Dynamic Rule-Based Reasoning in Smart Environments

Degeler, Viktoriya

Dynamic Rule-Based Reasoning in Smart Environments

Viktoriya Degeler

rijksuniversiteit
groningen

ISBN: 978-90-367-7286-0
ISBN-Electronic: 978-90-367-7285-3

Printed by *PrintSupport4U* / www.printsupport4u.nl
Cover design by Elena Tarasova
Cover image ©itestro - Fotolia.com

rijksuniversiteit
groningen

Dynamic Rule-Based Reasoning in Smart Environments

Proefschrift

ter verkrijging van de graad van doctor aan de
Rijksuniversiteit Groningen
op gezag van de
rector magnificus prof. dr. E. Sterken
en volgens besluit van het College voor Promoties.

De openbare verdediging zal plaatsvinden op

maandag 29 september 2014 om 14.30 uur

door

Viktoriya Degeler

geboren op 1 augustus 1984
te Charkov, Oekraïne

Promotor
Prof. dr. M. Aiello

Copromotor
Dr. A. Lazovik

Beoordelingscommissie
Prof. dr. Krzysztof Apt
Prof. dr. Michael Beigl
Dr. Amedeo Cesta

To Ukraine,
my Motherland

Contents

Acknowledgements

The date of my defense comes closer. As I am sitting here in Cardiff, where a new chapter of my life recently started, it is fascinating to look back over another life chapter that is about to be completed, the time of the PhD studies in Groningen, the time that started with my first working day as a new PhD student of the Distributed Systems group, and ended four years and two months later, with the internal thesis defense tryout that marks the final preparations of a PhD student who is about to defend.

I am deeply grateful to my supervisors, Alexander Lazovik and Marco Aiello, who guided me though the fascinating, unparalleled and sometimes puzzling experience of the PhD research.

Alexander Lazovik has everything PhD students can hope for in their direct supervisor. When I am thinking about stating here all accounts of his help and guidance, I quickly give up and regard this as simply impossible to list. His help is invaluable in every result and every paper I had over the course of this PhD. He is always there to discuss a new research idea or critically look at existing ones. Even when I was ready to throw away parts of my work as useless, he was the one who could spend hours, discussing with me the applicability of them, being able to see many points of interest I overlooked, and reigniting my passion to continue the research.

Marco Aiello is always ready to give a wise guidance when further direction is unclear. His broad views and timely advice helped me many times to choose a better course of actions, and many times things would have been so much harder without his knowledge and experience. He always carefully considers not only the content of one's own research, but also its place within the whole research community. I have learned much from Marco.

Research is almost never done alone. I would like to thank everyone who shared parts of this journey with me. I was lucky to have co-authors with extremely diverse knowledge. It is marvelous to see the results that a good collaboration can bring. Over the years of my PhD studies, my co-authors included Alexander Lazovik, Marco Aiello, Ilče Georgievski, Tuan Anh Nguyen, Andrea Pagani, Doina Bucur, Faris Nizamic, Francesco Leotta, Massimo Mecella, Rosario Contarino, Luis Ignacio Lopera Gonzalez, Mariano Leva, Paul Shrubsole, Silvia Bonomi, Oliver Amft, and Rix Groenboom.

Without any doubt, the research group of a PhD student has enormous impact on the quality of the time spent during their PhD years, and surely on the quality of the thesis itself. I consider myself very lucky to be a part of such lively and diverse group as is the Distributed Systems. I would like to thank all people from the group for being there and sharing our adventures together: on oberseminars, on Schiermonnikoog retreats, on group meetings, on all our off-hour activities. The group has had many members over the years: Mahir Can Doğanay, Elie El-Khoury, Eirini Kaldeli, Ando Emerencia, Andrea Pagani, Pavel Bulanov, Heerko Groefsema, Ehsan Ullah Warriach, Saleem Anwar, Ilče Georgievski, Tuan Anh Nguyen, Faris Nizamic, Fatimah al-Saif, Nick van Beest, Doina Bucur, Alexander Lazovik, and leading it, Marco Aiello. It amazes me to see how the group changes when a new person joins, always bringing a new unexpected sparkle with them. Or when a fellow PhD student moves on, taking away a part of the group's memory and character. And yet, I truly feel the connection with every person who has been a part of our group. I remember, Marco once said that if you have been a member of the group, you always remain a group member, wherever you are now. I guess, this is what they call a distributed system. I have had an amazing four years and two months in our everchanging group, and I want to thank each and every one person of the group for this! And to those who are now in the midst of their journey, good luck with the thesis writing!

An extra mention is reserved to my office mates, Eirini Kaldeli and Fatimah al-Saif. I remember our conversations, exchange of ideas, our laughs and even silent times full of work immersion. I appreciate Eirini for being an older PhD mate and inviting me to be a paranymph at her defense. I am amazed by openness and cheerfulness of Fatimah.

I would like to respectfully thank the reading committee of the thesis, Krzysztof Apt, Michael Beigl, and Amedeo Cesta, for their time and efforts in evaluating this dissertation.

My special thanks goes to people who helped me in the final moments of the thesis preparation. I would like to thank Ando Emerencia for giving his time to

translate the abstract of the thesis to Dutch. My paranymphs Ilče Georgievski and Tuan Anh Nguyen not only shared much of research work and publications with me, but also agreed to stand with me on the front line of the defense, and I am truly grateful for it.

I want to thank to Esmee Elshof, Ineke Schelhaas and Desiree Hansen, our secretary team who provided the much needed administrative support and advice.

I remember with joy the time of my internship in the Sapienza University of Rome. I would like to thank Massimo Mecella for supervising my work there. He provides a great guidance and support to his own PhD students, and I was happy to be a part of his group during my time in the University of Rome. I would like to thank Francesco Leotta for our fruitful collaboration that resulted in a joint publication. And also to all the people who made my time in Rome brighter: Riccardo De Masellis, Alessandro Russo, Andrea Marrella, Donatella Firmani, Marco Ruzzi, Jonas Neivelt.

There were many more who played important roles in my life during the times of living and working in Groningen, in the University and outside of it. It is almost impossible to mention everyone, yet all have my memories and my thanks for making the life more interesting and full: Aree Witoelar, Kerstin Bunte, Petra Schneider, Ioannis Giotis, George Azzopardi, Charmaine Borg, Ahmad Waqas Kamal, Dan Tofan, Ben Lewis Evans, Alzbeta Talarovicova, Elena Lazovik, Alexander Solovyov, Oleg Pidsadnyi, Olena Palaguta, Antoni Gostynski, Dmitrijs Milajevs, Nick Ruiz, Jenn Ruiz, Renato Higa, Ralf van den Broek, as well as everyone from our Improv group and Ralf's boardgames gatherings.

The unfortunate consequence of doing your work in a foreign country is the necessity of being away from your family. But no matter where I am, I always know that I have support of my parents, Oleksandr Tarasov and Valentyna Tarasova, and I feel that they are never too far away. My father Oleksandr is the one who first showed me the mesmerizing world of programming and computer science. He is also the one responsible for my earliest hacking experience, when I taught myself to break computer passwords to be able to play videogames outside of allowed hours at the age of nine. My mother Valentyna supported me in many adventurous ideas, and it is extremely reassuring to know that I can always talk to her during harder hours of my life, and receive support, advice, and understanding.

With my dear sister, Elena Tarasova, we not only shared the time of our many games as children, but also remain close now. She is a person with whom I can always share the joys and sorrows. I look with delight at her many artist skills, and it is always very exciting to see how the mathematical background of mine complements the artistic one of hers. She is the one who designed the cover of this

thesis.

It is customary to finish the acknowledgements with the one's closest person, and I am not the one to break this rule. In my heart and in my life, Andrii, my dear husband, takes the first and the greatest place. We came to Groningen together, and together we lived the Groningen chapter, sometimes joyous, sometimes rough and sad, sometimes rewarding, sometimes difficult. Yet difficulties we overcame. I cannot imagine doing this PhD without Andrii's support. And at the end of every our chapter, as always, we encounter these words: To be continued...

Viktoriya Degeler
Cardiff
August 25, 2014

Chapter 1

Introduction

On one winter morning a family is woken up by the alarm clock. The house is already prepared by the Smart Home system. The boiler that was turned off for the night, started heating water about 15 minutes prior to this, and the coffee machine is turned on to prepare a cup of coffee.

As it is an early winter morning, it is still pitch black outside when inhabitants wake up, therefore the artificial lights are turned on in rooms with someone inside to provide enough light for people to do their morning tasks. But while inhabitants are busy with their usual morning routine, it is gradually getting brighter outside. The artificial lights are gradually dimmed in response, so that the total light level stays the same over time.

Barbara sits to read her morning newspaper, and the desk lamp is automatically turned on to provide optimal reading conditions. David starts to check his e-mails, therefore the artificial lights near the PC screen are dimmed and the window blinds are automatically closed to avoid reflections on the screen. Several minutes later David finishes his task and leaves the PC. There is a problem however, which David does not know about: a presence sensor that detects his presense in front of the screen started to behave unreliably lately. The sensor readouts still claim that David sits in front of the PC. Thankfully, the Smart Home system can use the readouts of other sensors to verify the situation. Other sensors show the absense of the pressure on the chair, no typing sounds, or PC controls manipulations. Even though the situation is ambiguous for the Smart Home, the calculated probability of David's leaving is higher as several other sensors all consistently support it. Therefore the system reacts by turning off the screen to conserve energy, and opening window blinds again. The reliability status of all contradicting sensors is lowered, but the presence sensor is being hit the most, as it contradicts to all other sensors that are consistent with each other As such situations already happened a couple of times, the Smart Home system generates a warning about the sensor, so that people will be able to call a technician or to change the unreliable sensor themselves.

After consulting inhabitants' agendas, the Smart Home system gets the time

when the last person should leave the house. The heating is turned off twenty minutes before this time, to produce the biggest energy saving without hindering occupants' comfort. However, the temperature and air quality conditions are monitored in real time, and if something suddenly changes, a person stays at home longer than planned, or the temperature is about to drop below comfortable levels earlier, the Smart Home reacts by recalculating the heating trajectory for new conditions in advance and turning the HVAC system back on. This goes largely unnoticed for people inside the house.

After everyone goes to work, a house is immediately put into slumber, with most of devices that may have been turned on earlier, such as a TV, a radio, lamps, a clock, being turned off immediately, as there is no one in the house to use them at this time. A dishwasher is turned on an hour later. When the dishwasher finishes its work, a washing machine is turned on. These devices could have been turned on at practically any other time, but during cold and dark winter mornings, when lots of people get ready to work, many devices are used simultaneosly across neighboring houses, and the morning energy prices are thus usually very high. By using devices with low dependency on surrounding conditions outside of busy hours, and only one after another, the Smart Home is able to get the cheapest energy price, as well as to use to their fullest potential cheap renewable energy from solar batteries that are installed on the roof and a small residential-grade wind turbine outside.

This thesis describes the work done towards the realization of such a smart home, and the presented scenario introduces the most important topics that will be discussed in the dissertation.

1.1 Reasoning in Smart Environments

Smart homes, and in general other types of smart environments, can be defined by several important characteristics. The most important is undoubtedly the ability to be context-aware, to sense the physical surroundings and to understand the context of the current situation. Also smart environments should be able to reason using this information and to deduce valuable knowledge. And finally, they should have the ability to act intelligently in response to changing situations, according to certain goal criteria. Smart environments are often ubiquitous, which means their sensing and acting capabilities come from devices that are embedded in the physical world.

There are several criteria according to which the intelligence of smart environments can be judged. Most smart environments are designed to increase the comfort and quality of life of their users, e.g. inhabitants of a building. The automation of surrounding devices usually goes towards this goal, for example by understanding current user goals and problems and performing actions directed towards solving it. In most of the cases, however, this should not lead to situations where users are unable to override system's decisions, as this not only severely decreases their comfort levels, but also may be dangerous in some unaccounted situations. Therefore the ability of users to control the smart environment is also an important criterion. Many smart environments are designed particularly to help elderly or disabled people, thus supporting healthy ageing. And, of course, increasing energy prices and adoption of renewable energy sources bring forth the topic of energy awareness and energy savings in smart environments.

Most of current commercial smart environment products present only partial solutions, such as automated lighting or energy awareness. Several factors that slow the commercialization of full-scale smart home solutions include the necessity to greatly fine-tune the solution to every new location, the integration and coordination efforts between different components, efforts to keep consistent model across sub-systems that come from different sources, and so on. To summarize, the great amount of efforts that are needed to transfer the solution from one location to another hinders the deployment streamlining possibilities. Therefore this thesis takes great efforts to produce solutions that are fault tolerant, are easy to evolve when new requirements appear, require minimum information and configuration to be useful, and which allow to reuse the solution in other similar situations with minimal or no changes. As examples, Chapter 3 investigates a pattern that most of smart home architectures inevitably follow, while Chapter 7 shows, how the scheduler, which was originally implemented for scheduling devices in order to minimize energy price, is easily reusable to schedule the deployment of services to cloud environment by only creating an additional interface to the module.

Next we discuss several important scientific challenges for current pervasive systems, smart homes in particular. We also present specific research questions (marked with "**RQ**") related to these challenges that were addressed in this dissertation.

Over twenty years have passed since the first context-aware project Active Badge [Want et al., 1992] was developed. Over these years many different context-aware projects took place and many unique smart environments were designed and constructed, such as those described in Section 2.1 of this thesis. It is important for new projects to gain maximum benefits from the knowledge of the former

projects. One of the areas where such benefits can be gained is knowledge reuse in architecture design, components construction, their communication and integration. However, it is challenging to discern if a similarity indeed represents a certain pattern in smart environments design or it is peculiar to only some of them. On this basis we present the first research question.

RQ1. *What are the commonalities in the design and development process of smart environments? Can any pattern be derived from technical architectures of such systems? How can the process be streamlined, made easier? Which knowledge from existing projects can be reused in new projects?*

One of the most important components of any smart environment system that has actuators is the reasoning module which finds the actions to be performed by the system in any moment in time. Depending on the project, reasoning engines can have different functional and non-functional requirements. Among the most common ones is the requirement to be scalable, so that the system can grow beoynd several devices within a single room up to a big multi-story building. High levels of fault tolerance and robustness are also required for any system that has to be operational on a 24/7 basis. The reasoning module should be able to return real-time responces to any changes in the environment which require immediate attention. Another important requirement for commercial success of smart homes is to be dynamically adaptable, which means the system should not require large reconfiguration efforts when a new device is added or the old one is changed or removed. Among the non-functional requirements the computationally efficiency should be mentioned, i.e. the ability of a reasoning engine to perform only the necessary minimum of computations. This requirement is strongly connected to the scalability requirement, since in a fast-changing environment with many events per second, unnecessary computations may severely slow down and stress the system.

While several domain-independent techniques, such as AI planning [Kaldeli et al., 2013], have been used to reason in smart environments, the proper utilization of a domain structure may help to greatly increase the performance of a reasoning engine. However, this must be balanced to avoid fine-tuning a reasoning engine so that it will require considerable additional efforts during redeployment in other smart environments, potentially of different types. This formulates the requirements for the second research question.

RQ2. *What is an effective approach to design a reasoning engine for smart environments that fulfills all important requirements (e.g. scalability, robustness, dynamic adaptation, computational efficiency, real-time response, and so on)? Is there any specific structure or some distinguished features of smart environment*

domains? If yes, can this specific structure be exploited to increase the performance and/or reasoning capabilities of a reasoning engine operating with such domains?

The reasoning in decisions making for smart systems is very vulnerable to errors and incompleteness of sensor data. If an incorrect sensor reading gives a wrong impression of the current situation, the corresponding actions are likely to be not the optimal ones, and may in some situations even be harmful. Therefore, a very important challenge for smart environments is to detect as many sensor errors as possible on all levels of the system. Some errors, however, cannot be confidently detected. In this case, obtained data may be partially conflicting. The third research question is intended to mitigate this problem.

***RQ3**. How can the effect of sensor errors be minimized with respect to decision making? Can a reasoning engine work with incomplete and/or conflicting sensor data? If there is no definite answer on which data is incorrect, can the system operate correctly in presence of conflicting data?*

Energy saving potential is one of the important benefits that smart environment systems help to achieve. Recent research [Kok et al., 2008; Taqqali and Abdulaziz, 2010] shows the shift towards differentiated energy prices, adoption of smart grids, and future possibilities for buildings to choose their own energy providers dynamically. If this is the case, how can a reasoning engine for a smart system use the information about energy providers and internal information about devices and their expected consumption, in order to achieve lower energy prices and energy consumption savings? This forms the final research question of this thesis.

***RQ4**. How can a smart system utilize the existence of diverse energy providers in order to minimize the cost of energy over time? Does this smart system affect total energy consumption? Which information should be available to a reasoning engine in such a case, and how to use it in the optimal way?*

These are the questions that are investigated in this dissertation. In particular, the question RQ1 is answered in Chapter 3, where the pattern of the general architecture of such a smart home system is described. The question RQ2 is answered in Chapter 4, which describes the Rule Maintenance Engine that handles real-time system's reactions to people's activities, such as turning on coffee machine just before the alarm time, turning lights, closing blinds when someone works with PC, etc. Chapters 5 and 6 give an answer to the question RQ3 by describing context consistency diagrams that can resolve ambiguous, conflicting and incomplete sensor data based on information from other sensors. Chapter 7 addresses to the question RQ4 by presenting the mechanism for devices' scheduling to provide the minimum energy consumption and price.

Most of the work in this thesis is done as a part of the GreenerBuildings European FP7 project, where several ideas on smart building systems have been researched, implemented and tested in real living lab conditions.

1.2 A case of a smart environment: the GreenerBuildings project

GreenerBuildings[1] is a European FP7 project that aims to create a smart automated environment that combines automation for user satisfaction with energy-efficient adaptation. As a part of the project, an intelligent office is constructed on the premises of the Eindhoven University of Technology, The Netherlands. The project allows its users (i.e. people within a building) to establish and modify the rules of the building's behavior, so that the system automatically adapts to their needs by using the context information. The project features advancements in many research areas, including wireless sensor networks, smart grids, activity recognition, thermo-fluid dynamics, etc.

The GreenerBuildings project architecture is shown in Figure 1.1. The architecture is divided into three layers: the Physical layer, the Ubiquitous layer and the Composition layer.

The Physical layer is responsible for handling the devices of the system, including sensors and actuators, and for the underlying low-level protocols. There are many types of devices, among those are Plugwise devices, KNX controllers for blinds and heating system, motion, light, CO_2, and temperature sensors, etc. Though devices are operated via different protocols, the Sensors and Actuators Gateway service collects all information from raw devices and presents it in a uniform manner to higher levels of the system. The Interconnection with Smart Grid service provides the ability to be aware of the external energy pricing, and internal vs. external energy availability (e.g. from an internal wind turbine vs. external energy providers). The awareness about energy supply helps the GreenerBuildings system to adjust its demand and reduce the energy costs of the building operation.

The Ubiquitous layer ensures the proper operation of the whole system. The Repository is the database of the system. It contains all information about devices, their configuration, states, available actions that are represented as web services, and energy consumption. It also logs historical information about environment state which can be retrieved later for detailed analysis. The Context component collects information from sensors and transforms it into a high-level domain know-

[1] `http://www.greenerbuildings.eu/`

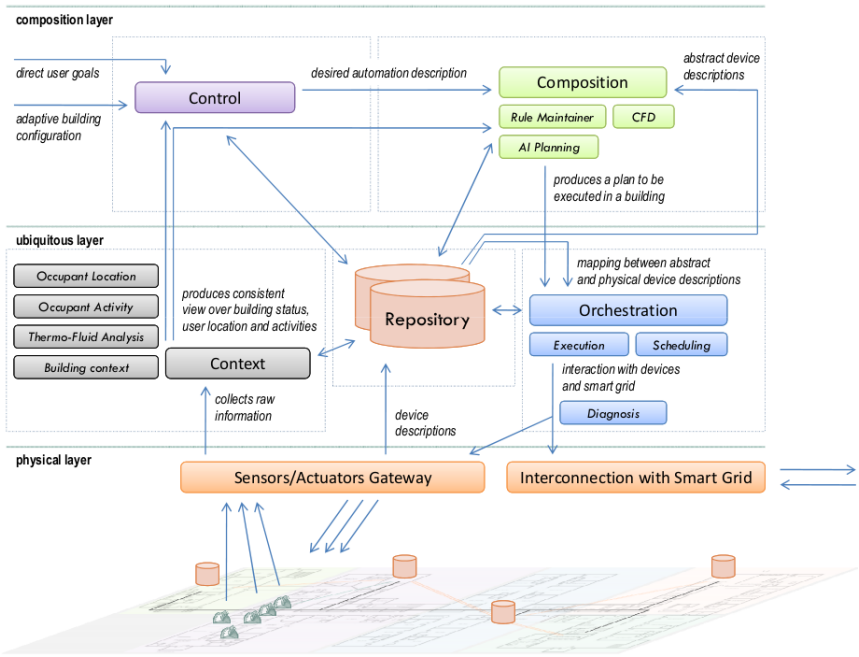


Figure 1.1: GreenerBuildings Architecture

ledge. This includes activity recognition, to represent such high-level activities as "a person is working with the computer", "there is a meeting/presentation in the room", etc. The Orchestration service is responsible for properly executing actions in a concurrent asynchronous way.

The Composition layer contains the Control service, which represents the system's interface to its users, including web interface, smartphone applications, dashboards, etc., and the Composition service, which is the main reasoning and decision making component of the system. The Composition, in turn, contains the Rule Maintenance Engine, which gathers information about user preferences, and decides which goal state the system should be transformed into to ensure the maximum user comfort and the minimum energy consumption; the Planner, which finds the actions to be executed to transform the system to this goal state, and the Computational Fluid Dynamics (CFD), which handles the heating part of the building, including HVAC, air quality, etc.

1.3 Thesis Scope and Organization

The main goal of this thesis is to design and develop a constraint-based reasoning module for smart environments. The thesis starts by describing an architecture pattern for smart environments that shows, where and how the reasoning module should fit in the full system. Then the thesis describes two reasoning modules: one that immediately reacts on environment changes, and the other to schedule the work of devices for the future. The correctness of sensor information for the first type of a reasoning module is very important for correct decisions, therefore the thesis also describes a way to detect and probabilistically resolve sensor errors.

Chapter 2 gives an overview of other works that are related to this thesis. The chapter starts by describing important existing and past smart environment projects and their main goals and achievements. Then the chapter moves on to context-aware systems and describes how they have appeared and their most important progress landmarks. The issue of inconsistencies within context data is the next topic of the chapter. It will present different works done to mitigate the errors in sensor data and consequences of inconsistent data. The constraint satisfaction techniques that have been used in other smart environment research before are described next. The chapter especially describes other approaches to dynamic constraint satisfaction problem, as this is the problem that CSP engines for smart environments face. And finally, the chapter describes different usage of scheduling techniques in smart environments.

Chapter 3 of the thesis analyses existing and past smart home systems and projects. As will be seen from this chapter, many common patterns emerge in architectures or during the construction of such systems. Even though many projects design architectures from scratch, the knowledge of past projects can be reused to make architecture design easier, or, in some cases, a project can completely reuse existing architecture solutions.

It is noted in the chapter that a layered approach often works best for smart environments architectures. The reason for this is that there are several natural layers in such systems. The physical layer that contains physical devices and the protocols to communicate with them inevitably exists in every smart environment, as the main feature of such an environment is to be able to sense and interact with the outside world. The ubiquitous middleware layer contains the knowledge base of the system, and the software that is required in order to intelligently process the sensor information and issue proper commands to actuators. The reasoning layer contains the intelligence of the system, whether these are learning capabilities, data

analysis, decision making about the proper actions to react on changes in physical world, etc. Finally, most systems contain dedicated software that is able to interact with people directly, and such software is a part of the user layer.

When analysing architectures of existing projects, it can be seen that these layers are often recreated in similar ways in many of them, and contain similar sets of components. The main contribution of the chapter is therefore the creation of an architectural pattern for smart environments. The pattern describes the common layers that most smart home projects have and their interconnections. Then the chapter goes deeper into each layer, and discusses common components that can be contained within these layers. There are some essential components, which inevitably exist in almost all smart home projects, but other specific components may appear or dissappear depending on the exact needs of a project. For example, projects that mind energy saving will often contain capabilities to measure the energy consumption. Other projects may deal specifically with certain types of interfaces, such as a Brain-Computer Interface [Aloise et al., 2011], and therefore will contain a corresponding component as well. The chapter describes the essential as well as some of the most common optional components, and their place in the big picture of the full project architecture. The chapter finishes with some examples of existing projects and how their architectures fit into the described pattern.

Chapter 3 shows the necessity of a proper reasoning component for a smart environment. **Chapter 4** goes into details to describe a reasoning component that is based on dynamic constraint satisfaction principles. The chapter shows why the constraint satisfaction model is good for modelling decision making in smart environments that must react to changing environmental conditions. The chapter also shows that straightforward representation of the constraint satisfaction problem leads to a great number of excessive calculations. The main contribution of the chapter is therefore to propose a method of modelling the task as a constraint satisfaction problem in a way that avoids unnecessary recalculations with new events in the environment.

The intuition that most of sensor events, as well as most of environmental behavior rules (that act as constraints), affect only a small part of the environment has led to the design of a method that can detect the affected parts of the environment for every event, given current rules. The *dependency graph* data structure, which is a bipartite graph between constraints and actuators, contains information about potential interdependency between them. The dependency graph is a dynamic data structure that changes with time and shows, whether a constraint affects current environment (in this case the constraint is *active*), or not. The chapter also shows that by finding unaffected parts of the environment, these parts may be

removed from the next constraint satisfaction task, and even affected parts may sometimes be split into several independent subparts. This leads to much smaller solution space for every consecutive CSP task, thus much bigger scalability of the system. The chapter creates a formal model of the dependency graph representation, and formally proves that partial recalculation of affected parts still keeps the full environment globally satisfied and globally optimal.

The evaluation part of the chapter describes the development of the appropriate system module, and its usage in the real living lab environment. The evaluation shows that for all environmental events recalculations took just a few milliseconds. The chapter further extends performance analysis by performing simulations with different conditions that may affect the performance of the system. The dependency graph approach is shown to consistently outperform the straightforward CSP representation. The experiments also showed that the *clusterization* of the environment has a noticeable effect on the performance. The clusterization parameter shows how pronounced are the clusters of variables within the full environment, i.e. the subsets of variables, which are highly interdependent, but only loosely dependent on variables outside of the cluster.

Most kinds of reasoning in smart systems, such as the one described in Chapter 4, are susceptible to making incorrect decisions due to erroneous input information, e.g. the one from faulty sensor readings. Therefore **Chapter 5** describes a way to detect incorrect sensor readings in probabilistic manner by using interdependency rules between sensor variables and a *context consistency diagram* data structure as a way to find the most probable situation of the current environment, when sensor readings give ambiguous, incomplete, or conflicting information.

The chapter defines the notion of a *context*, the partial extended interpretation of a current situation based on a sensor reading by using interdependency rules between sensors. If sensors are interdependent, a value of one of them may tell us something about possible values of another one. This idea is used to combine sensor readings to form several *interpretations* of an environment. Extended contexts are combined in a context consistency diagram. This diagram is a directed acyclic graph that shows which contexts are consistent with each other and support each other, i.e. they all may imply the same actual situation in the environment; and which contexts are conflicting, i.e. there is no situation that is consistent with all these contexts.

In case when sensor errors are absent and there is enough information to sense the whole environments, the context consistency diagram (CCD) has only one leaf node, which describes the most probable situation. In case there are sensor errors, or there is not enough information available, the CCD may have several leaves, each

of them representing a possible situation. The chapter describes, how the CCD can be utilized in order to calculate probabilities of every possible situation. By calculating these probabilities, several questions may be answered at any moment in time, such as: (i) what is the most probable current situation? (ii) What is the most probable value of a certain sensor? (iii) Assuming that one sensor has a certain value, what is the probability distribution of values of another sensor?

The contributions of this chapter are several. The chapter formally defines the CCD and related notions. The properties of a CCD are defined and proven. Then the chapter shows, how the probabilities of situations can be calculated by using CCD. The chapter also describes, how the CCD can be efficiently maintained in real-time, and provides associated algorithms.

The evaluation is done in a living lab that contains two working rooms and a coffee corner. 19 sensors were used in total, together with three days of sensor readings. 11 rules were defined to capture interconnection between variables. Results showed that using CCD resolves more than 40% of errors, and improves the correct situation detection rate by up to 11%.

While CCD, that is described in Chapter 5, provides extensive querying possibilities to the interpretation of the environment with possible sensor errors, sometimes it requires extensive computations, thus limiting the size of every single CCD that can be created within an environment, though still allowing multiple CCDs to be present in the same environment (in this case variables in different CCDs are regarded as independent). Therefore **Chapter 6** describes a way to reduce a classic CCD, while still keeping the possibility to calculate the most probable situation. The *reduced context consistency diagram* (RCCD) removes intermediate nodes that contain common child, therefore severely decreasing the size of an original CCD, as well as the time to update it. The drawback being the reduced querying capabilities comparing to the original full CCD.

The chapter provides a formal definition of the reduced context consistency diagram, formalizes its properties together with prodiving respective proofs, and provides algorithms to maintain the CCD in real-time, and to unfold a reduced CCD into a full one.

The chapter also presents evaluation based on a working desk with six sensors and five dependency rules. The evaluation showed that RCCD resolved 49% of errors in the input data, and is therefore consistent with the evaluation on a different living lab setup described in Chapter 5. The chapter also provides performance characteristics of the RCCD, and compares the RCCD with the original CCD.

Chapter 4 describes the system to react to sudden changes in the environment in

real-time in energy-efficient manner. But some devices have loose dependency on the outside conditions, while consuming considerable amount of energy. Examples of such devices are fridge or boiler. To cater for such a case, **Chapter 7** deals with scheduling of devices over time for houses that are connected to a smart grid.

The smart grid assumes the limited amounts of cheap energy, with increasing price when bigger amounts of energy are needed at once. The cheapest energy usually comes from internal renewable sources, such as a solar panel or a small wind turbine. When more energy is needed than these devices are able to provide, the energy may be bought from different energy providers, which may be a neighboring house or a large industrial-scale energy provider. The prices from energy providers usually change over time, due to higher demand at peak hours (morning and evening hours), and lower demand at out-of-peak hours.

In such a setting, when cheap energy is limited and the price changes over time, it is important to schedule devices that are largely independent from human interaction in a way that reduces simultaneous usage and mostly shifts the work of devices to off-peak hours. Chapter 7 presents the algorithm to do exactly this.

Each device is given associated *policy* of operation. The policy describes constraints over the work of devices. For example, a fridge must be turned on for at least several minutes every hour in order to keep the temperature inside always cold enough. A laptop, in order to fully charge, must be turned on for a certain total amount of time, but the exact time is irrelevant, etc. The chapter describes several such types of policies that devices may have.

Then the chapter formally defines an optimization problem, that uses changing smart grid prices and policies of devices in order to create a schedule that obeys all policy constraints, and keeps the energy price at minimum.

The chapter describes the Scheduler, the module that solves this optimization problem. For every policy the chapter describes additional optimization checks that reduce the search space of a problem. The two types of checks are a feasibility check, which shows is a partial schedule still satisfies the policy constraint, and an alternatives check, which defines, if another partial schedule which produces similar symmetrical results has been already checked.

The evaluation was done in a living lab that contains two working rooms, a printer corner and a coffee corner. Six devices were a part of the evaluation: a projector, a laptop that must be charged, a fridge, a boiler, a microwave, and a printer for batch jobs.

The evaluation was carried out during four weeks. First two weeks no scheduling was done, and the information about unscheduled energy consumption and price was collected. At the third and the fourth week the Scheduling was carried

out. The chapter contains results from economic, energy saving, and performance perspectives. The results showed savings in energy price up to 50%, and savings in energy consumption up to 15% for scheduled periods.

The Scheduler module was constructed in a way to be domain-independent and reusable for other domains that contain similar policies. Therefore the module was reused for the setting of services deployment to a cloud environment. The chapter contains the description of how the module can be used in such a setting.

Finally, **Chapter 8** summarizes the work done in the thesis, and presents some general conclusions and reflections for future work.

1.4 Publications

The content of this thesis has been published in several scientific venues. The work is done in collaboration with various people, in particular Alexander Lazovik, Marco Aiello, Tuan Anh Nguyen, Ilche Georgievski, Giuliano Andrea Pagani, Faris Nizamic, Luis Ignacio Lopera Gonzales, Mariano Leva, Paul Shrubsole, Silvia Bonomi, Oliver Amft, Rosario Contarino, Doina Bucur, and Rix Groenboom. Table 1.1 gives an overview of the respective publications.

Chapter	Venue	Citation	Notes
1	IEEE International Conference on Service Oriented Computing and Applications (SOCA)	[Degeler et al., 2013]	GreenerBuildings project description
3	Creating Personal, Social and Urban Awareness through Pervasive Computing	[Degeler and Lazovik, 2013*a*]	Full chapter
4	IEEE International Conference on Tools wth Artificial Intelligence	[Degeler and Lazovik, 2013*b*]	Main chapter content. Best Student Paper Award
	IEEE Context Modeling and Reasoning (CoMoRea)	[Degeler and Lazovik, 2012*a*]	
5	IEEE International Conference on Pervasive Computing and Communications (PerCom)	[Degeler and Lazovik, 2011]	Main chapter content
	IEEE International Symposium on Ubiquitous Intelligence and Autonomic Systems	[Nguyen et al., 2013]	Living lab evaluation
6	ICST Transactions on Ubiquitous Environments	[Degeler and Lazovik, 2012*b*]	Full chapter
7	IEEE Transactions on Smart Grid	[Georgievski et al., 2012]	Scheduler description. Main chapter content
	Scalable Computing: Practice and Experience (SCPE)	[Nizamic et al., 2012]	Scheduler interface for cloud resources deployment

Table 1.1: Overview of publications that are presented fully or partially within chapters of the thesis.

Chapter 2

Related Work

2.1 Smart Environments

In the last years many projects were dedicated to intelligent buildings automation. Exhaustive reviews of such projects are given in [Cook and Das, 2007; Nguyen and Aiello, 2013]. Here we present the history of smart environment projects that are the most relevant to the research questions of this thesis.

The conception of smart environment systems started with the Active Badge [Want et al., 1992] as early as in 1992. Though Active Badge is the most commonly cited as the beginning of context-aware computing area, it can be seen that the main part of the project was concerned with making the environment (particularly, stationary phones) smarter. Thus Active Badge is as well the first project that was concerned with the smart environments, and implemented them.

One of the earliest projects aimed at full building automation was MavHome[1], which started in 2000 [Youngblood et al., 2004]. The project was oriented at discovering patterns of device usage and occupants' behavior by utilizing several learning algorithms. The project produced many datasets of activities and sensor data, which were used to provide predictions on future usage. The conclusion of the MavHome project was also a starting point for the currently ongoing successful CASAS Smart Home project by the same university.

The iSpace project, which started as iDorm[2] in 2002 [Callaghan et al., 2004], features a room in a dormitory of the University of Essex (United Kingdom) campus fully equipped with sensors and actuators. The project uses full range of devices, featuring temperature, humidity, and light sensors, door locks, infrared sensors, video cameras, as well as HVAC system, motorized blinds, window openers, and light dimmers. The system can remember the user's habits and automatically adjust its behavior accordingly, so that explicit requests for actions from the user can be minimized, unless, of course, the user changes his or her habits.

[1] `http://ailab.wsu.edu/mavhome/`

[2] `http://cswww.essex.ac.uk/iieg/idorm.htm`

The SmartLab Research Laboratory[3] [López-de Ipiña et al., 2008] was constructed in 2006 to create a model of interactions between people and the context aware environment that surrounds them. This laboratory is used in several research projects, including Assistive Display, ubiClassRoom, and Eldercare.

The CASAS (Center for Advanced Studies in Adaptive Systems) Smart Home project[4] [Kusznir and Cook, 2010] started in 2008 and has since produced many publications both in academic press and in mass media. The smart environment for the project is a duplex apartment at the premises of the Washington State University. The apartment is equipped with a grid of sensors, including motion, temperature, and power meters. The project heavily relies on Artificial Intelligence techniques such as Machine Learning in order to automatically recognize patterns of occupants behaviour and automate the building to provide help and increase occupants' comfort.

As a part of the Smart Homes for All (SM4All) project[5] [Aiello et al., 2011] that also started in 2008, a smart apartment was constructed in Rome, Italy. The project implemented sophisticated AI planning techniques, which produce a set of actions to adapt the house to user needs in every possible situation. The breakthrough of the project was the application of the Brain-Computer Interface, a great help for many disabled people, which features the ability to read brain impulses of a smart home user and transform them into a certain desire about the smart home state, which in turn can be transformed into a set of actions for smart home actuators.

The e-Diana project[6] started in 2009 and was concerned with creation of a unified platform for all possible sub-systems of a smart building, such as security, lighting, power consumption, HVAC, etc. The project also aimed to improve energy consumption efficiency of such buildings and to provide better situation awareness for infrastructure owners.

The GreenerBuildings project[7] [Degeler et al., 2013] started in 2010 and implements the intelligent office, constructed on premises of the Technical University of Eindhoven, The Netherlands. The project features the ability of users to modify the rules of office's behavior, which will then automatically adapt itself to their needs based on the context information. The project gives special attention to such issues of smart solutions as fault tolerance and scalability, which are essential for realization of smart solutions on a large scale, given hundreds of separate offices

[3] http://www.smartlab.deusto.es/
[4] http://ailab.wsu.edu/casas/
[5] http://sm4all-project.eu/
[6] http://www.artemis-ediana.eu/
[7] http://greenerbuildings.eu/

per building, or thousands of smart homes within a combined smart city.

2010 is also the start year of the ThinkHome project [Reinisch et al., 2010], aimed at optimization of the energy efficiency while maintaining user comfort. The project uses knowledge ontologies for reasoning about the home states, and plans to provide a comprehensive knowledge base for evaluation of control strategies based on relevant building data.

There are also many specialized projects, for example EnPROVE[8] [Neves-Silva et al., 2010] or BeyWatch[9] [Perdikeas et al., 2011] that mostly deal with energy saving part of the smart environments, however, we mentioned here only some of the general broad-purpose context-aware smart environments.

2.2 Context Awareness

Context-aware research includes all applications that can discover relevant information about the surroundings and base their decisions on it. Traditionally most context-aware research includes ambient physical devices or mobile personal devices, though strictly speaking, context-awareness may be applicable also to a purely virtual world. The first most influential vision description of context-aware applications is given in [Weiser, 1991]. The work also introduced the commonly used now term "ubiquitous computing", which describes smart devices and appliances that are integrated into the everyday life. Users do not have to think consciously when interacting with those devices, focusing on their own tasks instead.

The already mentioned Active Badge system [Want et al., 1992; Schilit et al., 1994] was the first successful context-aware application. The personal badge with infrared transmitter was used to localize people within a building, forward calls to them, perform automatic identification, etc. Years later and severely transformed for commercial usage, similar systems can be seen in multi-purpose smart cards, such as Octopus smart cards in Hong Kong [Pelletier et al., 2011].

Though originally in [Schilit et al., 1994] context-awareness was mostly treated as location awareness, further works gradually extended the notion of a context. Almost simultaneously several works [Brown et al., 1997; Ryan et al., 1998; Franklin and Flaschbart, 1998; Rodden et al., 1998] proposed to generalize context to include more and more additional parameters, such as time, weather conditions, surrounding people's identification, and other aspects of the environment.

[8] http://www.beywatch.eu/
[9] http://www.enprove.eu/

This trend culminated in the seminal work [Abowd et al., 1999], which stated that context should include any information relevant to the particular situation or entity. Particularly, they stated that *"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."* According to them, the context-aware application is then defined as follows. *"A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the users task."*

In [Dey et al., 2001] a deep analysis of a conceptual framework for context-aware applications is proposed. The work discusses the main functions of such applications, the important requirements and abstraction, such as context storage and resource discovery, and finally, presents the implementation of a context framework, the Context Toolkit.

Since context awareness is such a broad and extensive research area, numerous surveys of the related research have been published over the years. Overview of pre-XXI century context-aware mobile computing research and the list of the most important implemented applications is given in [Chen et al., 2000]. More recently, a thorough review of different approaches to context-aware systems is presented in [Baldauf et al., 2007]. The work summarizes different context design and modelling techniques. It also mentions the concept of a layered architecture framework, which is similar to the layered architecture pattern that is given the full attention in Chapter 3 of this thesis.

Since by nature, the context information must be gathered from different places and by different types of sensors, well-agreed interfaces and protocols are essential for success of large-scale context-aware applications. Therefore, in [Truong and Dustdar, 2009] the usage of web services to enhance such applications is surveyed.

Over the years of context-aware systems research many different ways to model the domain-level information were devised, some general, some more specific to a particular task that the system was designed to solve. Among the most known high-level context representations are Resource Description Framework (RDF) [Lassila et al., 1998], W4 (Who, What, Where, When) Context Model [Castelli et al., 2006], the RDF-based Web Ontology Language (OWL) [Antoniou and van Harmelen, 2009], and Context Modelling Language (CML) [Bettini et al., 2010]. An extensive survey of different context representation models is presented in [Bettini et al., 2010].

Ontological representation with OWL language is particularly suited for web setting, and enables different context applications to communicate using the stand-

ardized context representation. For example, [Zhang et al., 2005] presents a layered Context Stack, which uses ontology on different levels to facilitate semantic context web sharing.

2.3 Context Inconsistency

Correct context determination is a crucial component of any serious pervasive system and has been extensively studied in the literature. In particular, various authors address the issue of inconsistent sensor readings for the problem of precise context determination.

The detection of contradictions in context based on predefined constraints is researched in [Xu and Cheung, 2005; Xu et al., 2006, 2010]. The authors propose to convert each constraint into a tree with constraint operators as vertices and contexts as edges. Then they introduce a partial constraint checking algorithm that is capable of re-checking only those parts of the constraint tree that may be affected by a new context. The work is extended in [Huang et al., 2008], where authors propose to check branches of the tree with probabilities. This enables fast processing of large trees and adds scalability to partial constraint checking, however it reduces the percentage of correctly found inconsistencies. The papers aim at fast detection of context contradictions, but do not concentrate on the problem of context interpretation when inconsistencies (contradictions) are present.

In [Bu, Gu, Tao, Li, Chen and Lu, 2006; Bu, Chen, Li, Tao and Lu, 2006] the context reasoning is performed by modeling the context ontology and then finding inconsistencies using ontological reasoning. The context is modelled as RDF-triples using OWL-lite language. They also present a context lifecycle, where new context starts at the "beginning" phase, can be "updated" during its lifetime, stagnate at the "inert" phase and finish its life as "disappearing". In the presence of a conflict they propose to discard one of the conflicting contexts based on their relative frequencies. Other techniques for resolving inconsistencies are proposed in [Xu et al., 2008]. They define several possible resolution strategies, among which are drop-latest, drop-all, drop-random, and drop-bad. The latter heuristic counts the number of conflicts for each context and drops the one with the biggest number. While those techniques can be used to successfully resolve straightforward inconsistencies, they may lead to retaining an incorrect interpretation in cases when proposed heuristics cannot confidently resolve the conflict.

In [Henricksen and Indulska, 2004] context properties are classified and initial ideas on handling several inconsistencies are outlined. While introducing a classification, they do not provide precise algorithms for dealing with possible context

inconsistencies. In [Lu et al., 2008] a mechanism for detecting failures in context-aware applications is provided, as well as means to test such applications. In [Huang et al., 2009] the detection of inconsistencies that may emerge due to asynchronous arrival of concurrent events is investigated. The proposed algorithm detects the original order of concurrent events based on the *happen-before* relation.

In [Kong et al., 2009] authors propose to extend the OWL ontology with fuzzy membership to tolerate inconsistencies. Their proposal involves manual assignment of membership values and does not propose a way to retrieve useful information from it.

A similar fuzzy approach is discussed in [Marcelloni and Aksit, 2001]. The authors try to minimize the impact of early incorrect decisions made during software design. They show that wrong classification of an entity to one of the mutually exclusive classes, if done early, may lead to further incorrect or suboptimal design of the software system. They propose to improve the process by deferring decisions about entity's classification as long as possible, instead of assigning fuzzy membership values to each of possible classes. However, the solution is not applicable to context reasoning, as it is based on human decisions about entity's properties membership values that have to be updated with each information change. This is acceptable for prolonged and slow software development process, but impossible in highly dynamic automated context-aware systems.

Context lattice [Ye and Dobson, 2010] is a data structure similar to the context consistency diagram in Chapter 5. Nodes of the lattice are logical context predicates, such as *inLivingRoom* or *remoteControlAccessed*, and they can be combined to create an intersection of predicates. Conflicting predicates all combine into a single FALSE node. The context lattice can be used on semantical level to show which contexts are compatible and derive more high-level semantics.

Similarly, [Mittal et al., 2012] propose to derive probabilistic association rules by using a *concept lattice*, which combines a set of contexts to represent a situation that matches the description of those contexts.

2.4 Constraint Satisfaction in Smart Environments

In the field of smart environments, several studies propose to use constraint satisfaction techniques to solve reasoning problems. For example, multi-agent coordination in smart homes is modelled as distributed constraint optimization problem in [Pecora and Cesta, 2007]. The coordination is fully distributed, i.e. every agent relies only on communication with other agents and manages one or more variables. In this scenario constraints model the desired minimum-cost concurrent behavior

of agents.

On the other hand, [Petersen et al., 2013] propose a solution with centralized command post for mission-critical environments, such as search and rescue operations with robot teams. They present a method for efficient task assignment, where constraints that are added by humans via a dedicated interface are combined with physical constraints of the environment, and are solved by a modern Mixed Integer Linear Programming (MILP) solver.

In [Koes et al., 2006] authors present a first order logic constraint language for such search and rescue domains for robots. They also introduce a goal-oriented Constraint Optimization Coordination Architecture (COCOA), which aims to transform the original problem by formulating it as a constraint optimization problem. The solution to this problem will generate a schedule that can be executed by a robot with some level of abstraction.

In [Cesta et al., 2001] a problem solving environment is described that deals with complex scheduling problems, which are represented as constraints. They present O-OSCAR, a CSP-based object-oriented scheduling framework.

CSP-based AI planner is used in [Kaldeli et al., 2010, 2013] to compose services for smart home scenarios. The planner allows the expression of extended goals and uses the latest advancements in the CSP field to make the search faster using enhanced inference techniques.

2.5 Dynamic Constraint Satisfaction

The formulation of the dynamic constraint satisfaction problem (DCSP) as a set of successive static CSPs with addition or removal of constraints was first proposed in [Dechter and Dechter, 1988], and subsequently elaborated in many other works.

In [Bessiere, 1991] the application of arc-consistency algorithms for Dynamic CSPs is investigated. Bessiere considers binary constraints, i.e. those that involve only two variables. The original arc-consistency algorithms do not solve static CSP completely, but eliminate all values that are mutually inconsistent and are definitely not part of the solution, but that otherwise would be discovered by backtracking procedures over and over [Mackworth, 1977; Dechter and Pearl, 1987]. For the Dynamic CSP the original arc-consistency cannot be reused if a constraint is relaxed, because the reasons for marking values inconsistent are not being tracked, and inconsistency marking cannot be removed without fully rerunning the algorithm. Bessiere therefore proposed an extention to the algorithm to track the original reasons for marking inconsistent values, which allows to make incremental changes to arc-consistency values when constraints are changed. An improved algorithm

with lower space complexity is proposed in [Debruyne, 1996].

Algorithms based on nogood recording that may be used in both static and dynamic CSPs are proposed in [Schiex and Verfaillie, 1994; Verfaillie and Schiex, 1994]. Similarly to arc-consistency, a *nogood* is a pair of value assignments that cannot be contained in any solution of the CSP. A set of nogoods is built during a backtrack search.

An algorithm to solve each CSP in a sequence of consecutive CSPs by using previous solutions is proposed in [Roos et al., 2000]. To avoid big differences in successive solutions, which are often undesirable in practice, they propose a repair-based algorithm RB-AC, which performs a local search in the neighborhood of an infringed solution to find a new nearby solution which is the most similar to the old one. The results however show that repairing a solution may be much harder than creating a new one from scratch if several constraints are changed simultaneously. Therefore [Ran et al., 2002] propose an approximate algorithm that reduces the time complexity of a repair by relaxing the optimality requirement with respect to number of changes made.

An alternative definition of DCSP was formulated in [Mittal and Falkenhainer, 1990], where DCSP defines a single CSP with different additional sets of variables and constraints depending on variable values. In this dissertation we use the definition as stated in [Dechter and Dechter, 1988], when referring to the DCSP. A comprehensive survey of the DCSP related research is presented in [Verfaillie and Jussien, 2005].

2.6 Scheduling in Smart Environments

There is a number of works that investigate the usage of automated scheduling techniques for smart environments.

In [Kreucher et al., 2006] authors consider the problem of scheduling sensors to better detect and track "smart targets", i.e. those that can realize they are being under surveillance and react in order to conceal themselves. They consider active and passive mode of sensors, where active mode has much more efficient tracking characteristics, but is also much easier detectable by targets in question. They argue for the necessity of a non-myopic approach for such scheduling, the one that maximizes long-term benefit as opposed to maximizing immediate gains.

Timing constraints to specify scheduling of tasks for automonous search and rescue robots are investigated in [Petersen et al., 2013].

Scheduling for smart homes in order to conserve energy is a very important usage of scheduling methods, and recently several studies have appeared on this

topic.

A decision-support tool that residents can use to optimize their usage of energy is proposed in [Pedrasa et al., 2010]. The tool allows its users to define the benefits they get from every energy service, and then by calculating the energy cost of these services, schedules them in a way to maximize gains and minimize losses. They use particle swarm optimization algorithm to solve the optimization task. Though it does not produce optimal solution, it is known to produce good enough solutions within manageable time.

A smart meter with changing electricity prices is considered in [Xiong et al., 2011]. They consider two types of devices, "real-time" that may consume energy when they require it and "schedulable" that can be scheduled by the system to consume energy at a later time. Given such a setting, the system aims to reduce demand peaks.

Similarly, in [Du and Lu, 2011] authors propose to schedule appliances based on forecasts of energy price and consumption, and on specified objectives of users' comfort. They propose a two-part algorithm, where the first part creates a schedule for the day ahead based on forecasts and the second part makes real-time adjustments to it.

Chapter 3

Architecture pattern for context-aware smart environments

The ability of pervasive systems to perceive the context of the surrounding environment and act accordingly proves to be an enormously powerful tool for raising immediate users' satisfaction, helping them to increase their own awareness, and act in a more informed way. Recent years marked many smart environment solutions hitting the market and applying latest pervasive computing research advancements on an industrial scale.

Magnitude of context-aware smart spaces applications is enormous. On the one side such applications include telephones that redirect calls to the room where the recipient is currently located, e.g. the Active Badge system [Want et al., 1992], and simple coffee machines with the possibility to schedule the time of coffee preparation exactly to the time when you wake up. On the other side there are the whole building automation systems with complex rules of behavior and planning techniques that are just waiting for your wink to launch the complex artificial intelligence reasoning that will understand and fulfill your unvoiced demands.

Going even further, smart environments matter not only on the personal and the social scale, but on the bigger urban scale as well. Sometimes whole neighborhoods can be considered as smart spaces, as shown by many Smart Grid enabling projects [Georgievski et al., 2012; Capodieci et al., 2011]. By introducing small scale energy generating facilities, such as wind turbines or solar panels, it is possible for individual buildings to produce more energy than they consume at certain points in time. To avoid losing this precious energy (which becomes even more precious considering its "green" sustainable origin), peer-to-peer-like energy transfer connections are introduced between buildings, with full featured automated negotiation techniques that enable one building to sell excessive energy to another neighboring building. First field-testing projects, such as PowerMatching City project in the Netherlands [Bliek et al., 2010], which features 25 interconnected households, show that not only such energy comes with a cheaper price, but also the "transfer overhead" is severely reduced, as the average energy travel distance

is much shorter.

As can be seen, context-aware smart environments come in many different flavors and on many different scales, but the underlying idea remains the same: the system is aware of its context, i.e. the environment around, is able to act accordingly in an intelligent, predefined, learned, or automatically inferred way, and is able to communicate to its users, thus increasing their comfort and awareness level as well. Seng Loke in his book [Loke, 2006] defines this as three main elements of the context-aware pervasive system: sensing, thinking, acting.

In just a few years after the first introduction of smart environments, the topic became booming, and many projects both in research and in industry were dedicated specifically to advancements in this area. As happened in many other research fields where a big number of different research groups and industrial companies started to work separately on the same topic, in the context-aware environments area the problems that the groups face are to a large extent similar, and some of them were solved several times, sometimes in a similar manner.

One of such problems, and an important one, is the high-level architecture design of the smart context-aware systems. Since the beginning of the 2000s, many projects have been designing and implementing the smart environment systems from scratch. However, when looking post-factum at the architectures of these systems, one can notice many similarities among them. With the same basic structure, the biggest differences usually arise at the level of individual components, aimed to satisfy different functional requirements.

Naturally appeared the idea to unify the architecture design for such smart environments projects. Taking many successful and undergoing projects as case-studies, we tried to find the common structure, the common patterns, and in some sense the "best practices" that can help future projects to reduce the efforts spent on the general system frame, and redirect those efforts to more specific requirements that are unique in every project. The work of Preuveneers and Novais surveys similar efforts to find and study best practices on different levels of smart pervasive applications that were already done in previous studies, including requirements engineering, context modelling, development acceleration and code reuse. In this chapter, on the other hand, we focus on a pattern for architecture design of smart environment systems [Preuveneers and Novais, 2012].

We will introduce several layers of the architecture that inevitably exist in one form or another, and discuss the possible components that may be parts of these layers. We will then discuss the common information flows within such architecture and mention the most notable problems, such as scalability and fault tolerance. Finally, we will present several case studies, successful or undergoing

smart building projects, and show that the presented pattern can be easily mapped to their architectures.

3.1 Architecture Overview

In this section, we present the design pattern of the smart environment architecture. The overview of the pattern is shown in Figure 3.1. We split the full architecture into four layers, with several distinct components in every layer. Most component patterns arise from the architecture design similarities due to requirements that are common for all context-aware smart environments. It is important to note that components are not exhaustive in terms of components' availability to the system. The components here are the backbone, but it is often the case that the actual implementation dictates for some support components, which either establish communication between other components, act as watchdogs, proxies, monitors, or solve other complementary functions. Also, if the system features a certain specific ability, such as a specific handling of heating mechanisms, or a special support for disabled users, more often than not this will require a separate component. Thus the system that is described in this chapter should be viewed as extensible, with the ability to plug-in more components, if needed. And, to the opposite, some presented components and flows are sometimes simplified, combined, or removed altogether in projects of smaller scale. This possibility will be highlighted at the level of components.

The Physical layer contains all hardware parts of the system, which include all wired and wireless sensors, actuators, network topology, low-level protocols associated with them, etc. One of the main tasks of the Physical layer is to collect information about the environment and transfer it to higher layers. Low-level protocols may be implemented to provide a common gateway, which allows to unify interfaces, hide the specific hardware differences, and/or reduce bandwidth requirements by bundling the information. The second main task of the Physical layer is to invoke actuators in the environment based on commands sent from higher layers.

The Ubiquitous layer acts as an intermediary to the system components, and has several distinct responsibilities. First of all, the layer contains system's data storage, which means it collects and stores all the current and historical information about the environment, system configuration, system capabilities, user preferences, etc. The layer also should contain an information processing component, capable of detecting simple sensor errors or faults, transform information from low-level sensor values to high-level logical state of the environment, and enrich sensor in-

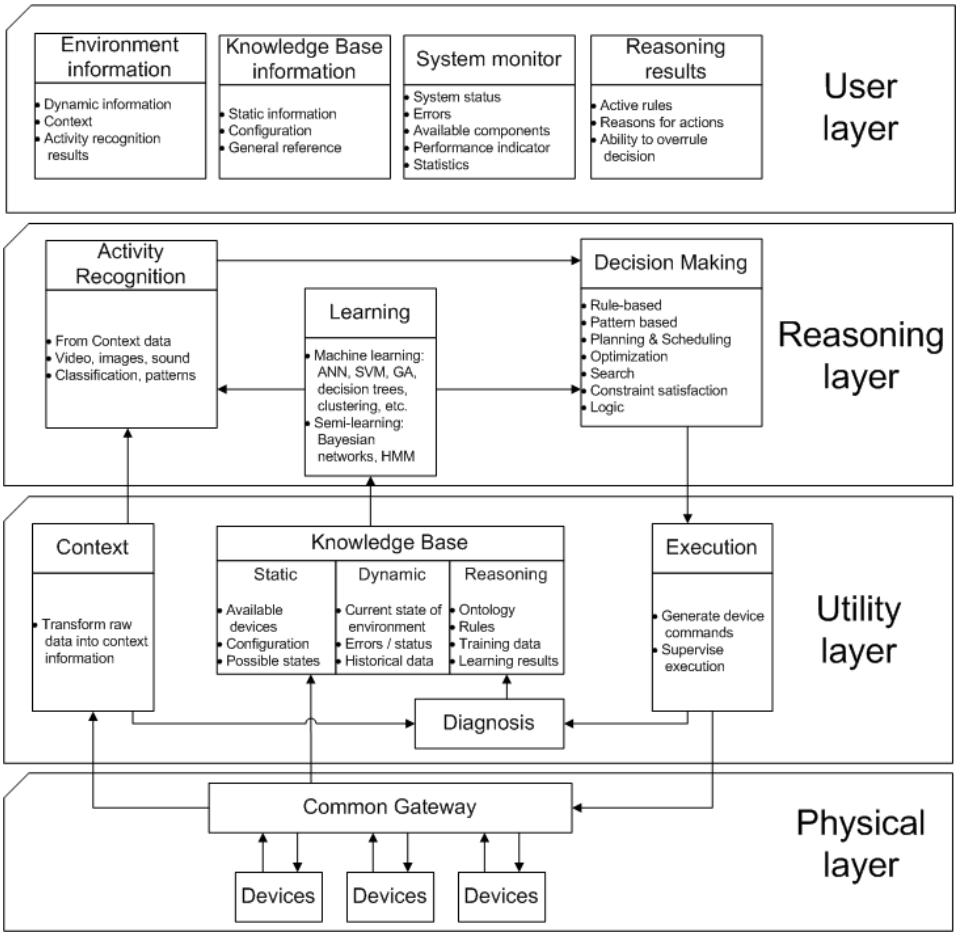


Figure 3.1: Smart System Architecture Pattern

formation. On the actuation side the Ubiquitous layer is responsible for transforming commands from the Reasoning layer to low-level commands that the Physical layer is capable to execute, and making sure they are passed to the Physical layer in a concurrent non-blocking way.

The Reasoning layer is the layer where system's logic resides. It contains all components that are responsible for decisions on system's actions, be it a simple logic defined through strict if-then rules, or sophisticated AI techniques, such as planning or scheduling actions. The layer may also contain activity recognition or learning components, which should improve the automated system's response.

The User layer presents information about the system to its users. It contains two main parts. The first one presents information about the environment, current user preferences, the reasons for certain decisions that the system makes, and allows a user to modify the configuration of the system according to her or his needs, enter new rules of execution, or overrule system's decisions. The second part provides meta-information about the system itself, such as the status of all components, whether they are working properly, statistics, and resources consumption.

We will now describe each layer in detail.

3.1.1 Physical Layer

The main part of the Physical layer is, as the name hints, physical, i.e. devices that are embedded in the environment. All groups that decide to implement a smart environment face an unavoidable issue from the very beginning: the heterogeneity of the devices they plan to use. Even now, while some companies started to specialize on providing combined sets of sensors and actuators, there are still many special devices tailored to a particular need with distinct and possibly proprietary interfaces and communication protocols.

Thus, it is essential to unify the interface and data gathering before sending the data further into the system. Not only such unification follows the famed low-coupling architecture principle, and makes it easy to add, remove, or change devices both individually, and as a whole type, but it also keeps all other parts of the system device-insensitive, so in its pure form the change of a device will not require a single line of code to be changed anywhere past the Physical layer.

The essential part of the Physical layer is the Gateway, the component that initially collects data from devices and applies low-level transformation to it in order to send it further into the system in a uniform way. Note that the Gateway is highly hardware dependent, and will usually require changes in case of any changes of device types.

Of course, conceptually some other parts can be also treated as physical devices, especially information-providing ones, such as person's agenda, a call event over VoIP, or some electronic message from outside the system. Often such events are initially processed and entered into the system from the "top", i.e. from the User layer, or even via some other distinct entrance point, directly into the Ubiquitous or the Reasoning layers. But we argue that with a good level of abstraction, which a well-implemented Gateway provides, adding such events to the system from the "bottom", i.e. from the Physical layer, is also a perfectly viable solution that serves well to the unification of the event processing and information flows. In this case such event generators are commonly viewed and regarded as virtual or logical sensors.

3.1.2 Ubiquitous Layer

The Ubiquitous layer is the backbone of the whole system, the main support of all other components. It also contains main channels of information flow and storage, and in some sense the layer connects and helps in the interpretation of two different worlds: the device-level Physical layer and the domain-abstracted Reasoning layer.

Knowledge Base

We start with the Knowledge Base component. The database of the system belongs to this component, and for some systems the component will also be synonymous to the database. However, there is much more to it, first and foremost with respect to the types of information it handles. There are at least three types of information that are usually stored in the Knowledge Base, Figure 3.1 summarizes them.

The first type is the static information about devices. This includes the types of devices the system has, their communication protocols, whether they are sensors or actuators, the structure of readings they provide or states they can be set into. For configurable devices it also contains the configuration information. Though the name "static" implies that the information does not change frequently, it is nevertheless possible that the information will change automatically during the course of system's operation. For example, the SM4All smart home environment provides an automatic device discovery feature [Aiello et al., 2011].

The second type of the Knowledge Base information is the dynamic one, and this represents the information that changes with high frequency, for example the current state of the environment, devices, or executed commands. Many systems prefer to send this type of information directly to relevant components (for example in the Reasoning layer) instead of sending it to the Knowledge Base, and letting

the Knowledge Base handle further distribution. This makes sense, since direct communication is also the fastest, and time of reaction is of utmost importance for the intelligent environment. However, the need for historical data collection is almost always a requirement, whether it is to update the training of a learning mechanism, to diagnose errors, or to show the history to a user. This means that even if the direct link for dynamic information transfer is outside the Knowledge Base, there should be a duplicate link which sends the data also to the Knowledge Base for storage and further retrieval and processing.

Finally, the last type of information in the Knowledge Base is almost exclusively used by the Reasoning layer, as it contains all the information, required for high-level reasoning. The exact model of information here depends heavily on what kind of reasoning the system uses. For ontology-based systems such as ThinkHome [Reinisch et al., 2010], this will be the ontology of the system and the environment. For rule-based behavior the reasoning rules will be stored. Training data and learning results will be present for all systems that use machine learning in one way or another.

Context

The next component of the Ubiquitous layer is the Context. The main task of the context is to transform low-level raw data gathered from devices into higher-level information suitable for the Reasoning layer.

One type of such processing is data packaging. Some sensors, for instance an acoustic sensor, send information with a very high frequency. It may be the case that the higher level components do not need such detailed information. Some simplified systems may only need to know if there is a sound or not or its volume, thus large amounts of data transfer may be avoided by combining the information on the Context level and only sending the results higher into the system. Not only the bandwidth is saved, but also it removes the need for the Reasoning level to have a lower level representation of the device, and allows it to think in "domain-level" terms. Other examples include simple error filters that work nicely for such sensors as motion or light sensors, which for the most part send correct readings, but may occasionally send faulty ones. Such outliers are easily detectable by comparing them with neighboring readings.

It should be pointed out here that the Context component in its pure form does not involve any kind of domain level reasoning, such as activity recognition. The Context instead must prepare the data for the high-level reasoning by abstracting some devices and transforming the data from other devices. As an example, let us take the presence detection. Though in its basic form it is a simple mapping with

the motion sensor, the recognition of presence in the room already reasons and operates in domain level terms. To increase the sophistication level, other sensors may be used in later versions of the system in order to get better recognition rate (such as RFIDs on entering people, video stream, etc.). This will change trivial mapping into intricate reasoning system. Thus from the beginning such reasoning should be placed into the Activity Recognition component of the Reasoning level.

Execution

The Execution component is in some sense the exact opposite to the Context. The task of the Execution is to transform action goals received from the Reasoning layer into executable actions that can be sent to devices. An important addition to the task is also to oversee the correct execution of the commands by devices.

It should be noted that the Execution in its pure form, as well as the Context, has absolutely no domain-level reasoning, i.e. it should not decide which command to execute out of several possibilities (any form of such reasoning belongs to the Reasoning layer). A good example is that the command to the Execution to "turn on a lamp" should also specify exactly which lamp should be turned on in case there are several of them. If, on the other hand, the command is general, as in "turn on anything that provides light", the Execution then also assumes some responsibilities of the Reasoning layer components as there may be several ways to satisfy the request (e.g. turning any one out of several available lamps in a room), and the Execution component must be able to choose one of these several available executions by using some criteria. In practice, it still may be a viable solution, in order to reduce the complexity or simplify the architecture, but the system architect in this case should always be aware of this mixing of responsibilities, understand the reasons for them, and evaluate alternatives.

Even with this being said, the Execution has (and must have) some form of reasoning on the level of particular devices. For example, it must be able to match the correct execution action with the desired end-state of the device. Also, if some command always involves actuation of several distinct devices in a uniform manner, such a command can be abstracted on the Reasoning layer to a single atomic action and only inside the Execution component it will be transformed into a series of commands applicable to each device.

Diagnosis

The last component of the Ubiquitous layer is the Diagnosis component. This component is optional, i.e. some systems choose not to implement it explicitly,

especially in the early stages of smart environment development.

The task of the component is to monitor readings from sensors and execution results, check the correctness of the devices, and detect any anomalies, if possible. For example, many battery-powered devices tend to send erratic data when the battery is low. This may cause large problems at the reasoning level, if not detected earlier.

The diagnosis may also have its counterpart at the reasoning level, which will use domain data together with the information from the Diagnosis to forbid the usage of faulty devices, until fixed, thus restricting the available domain.

3.1.3 Reasoning Layer

The Reasoning layer contains the domain-level logic of the system. This is the most diverse layer as well, as every smart environment project has its own ideas on how the environment should reason and make decisions about the actions it should perform.

The choice of the exact context representation model influences heavily the capabilities for system's learning, activity recognition, and decision making, thus it is among the most important choices to be done during the early design of the smart environment system. The Web Ontology Language (OWL) [Antoniou and van Harmelen, 2009] and the Resource Description Framework (RDF) [Lassila et al., 1998] are the most popular choice for context representation at the moment, but the possibilities are certainly not bound to them.

We split this layer into three main components. However, we note that smart environment projects can have any combination of these components or their subcomponents.

Learning

The Learning component is responsible for automatic learning of the best possible decisions and actions based on input data, which can either be a real-time data, or previously gathered training data.

The Learning component has a bit special place among all other components of the system. On the one hand, this component is optional, i.e. it is possible to construct a smart environment system without any learning incorporated, for example if it is a rule-based system. On the other hand, if the component exists, it takes one of the most important places in the system.

Machine learning methods are numerous: artificial neural networks, support vector machines, decision trees, genetic algorithms, reinforcement learning, differ-

ent clustering techniques, etc. They are all applicable for usage in smart environment systems.

Of course, when we speak about the learning capabilities of the system, usually it implies that the system has the ability to re-learn and re-train automatically when initial data changes, e.g. a user develops a new habit. However, there is also another possibility, a "semi-learning" system, so to say. In such a system the Learning component is not an integral part of the day-to-day system operation. Instead, the learning is performed using a standalone learning module at the beginning on some initial existing data, and results are entered to the system as unalterable rules. They are often represented by Bayesian networks or hidden Markov models. In such cases the Learning component may often be omitted from the operational architecture, as it indeed is not involved in the operational flows. When the need arises to relearn or retrain the system due to considerable changes in the outside world, the standalone learning module may be launched again, and the new operational rules will be entered to the system to replace the obsolete ones.

Activity Recognition

The Activity Recognition gets the information about the current state from the Context, and applies internal knowledge to classify and define more high-level information about the environment. For example, while the Context may send a reading from a motion sensor that there is motion in the room, the Activity Recognition will recognize that it corresponds to someones presence in the room. Given the stream of video from the Context, the Activity Recognition may define a whole set of the new domain-level information, such as whether a person is working with PC, thinking, eating, moving around, etc.

Theoretically this component is not obligatory, as it is possible to make decisions directly based on the information from the Context. However, without the Activity recognition the complexity of decisions is severely limited, as they lack a big part of high-level domain information.

The activity recognition may include sound, image, or video recognition. Often it uses results obtained from the Learning component in order to classify and recognize the activity. Sometimes activity recognition may contain stricter definitions of what a certain activity means (such as a certain state of sensors corresponds to a certain activity), in which case the recognition itself checks the correspondence of the definition to the current state of environment.

The results of the Activity Recognition component will go to the Decision Making component, where, combined with the information from the Context, will depict the full knowledge about the current state, which in turn will be used to make

decisions.

Decision Making

The Decision Making component is what turns the intelligent environment from a silent observant into a resolute actor: it decides which actions should be performed in a given situation with a given knowledge.

As with the Activity Recognition and the Learning components, the Decision Making component comes in many different forms, at least as many as there are fields in artificial intelligence and systems automation research areas. Some usable techniques include optimization theory, planning and scheduling, constraints satisfaction, search techniques, logical reasoning, ontological reasoning, reasoning under uncertainty, and many more.

The important difference to note is that decision making may be split into two types: instant and continual. Instant decision does not mean instant execution. However, it means that the decision, once it is made and sent to the Execution component, cannot be revisited and changed. Instead, the feedback from the environment (even if it is a feedback about errors in execution) goes to the "new cycle" of decision making, and requires new decisions to be made. The instant decision making is easier from the architectural point of view, particularly it goes well with stateless components, because every new decision can be made independently from previous ones.

However, sometimes instant decisions are not enough. Continual decision making usually involves several steps of execution within one decision. It also involves remembering the decision and revisiting it after receiving new feedback, possibly alternating some steps. Unlike instant decisions, continual ones usually require stateful components, thus are more demanding with respect to fault tolerance and general architectural cleanness.

3.1.4 User Layer

Though many projects opt not to give particular attention to interfacing with users, instead specifying UI as a part of some other architecture layer or component, we argue that it deserves a separate dedicated layer in the architecture.

The User layer provides a view of the system to the user, and, which is even more important, it gives the ability to change and fine-tune the system, to debug errors, to override system's decisions and much more.

In this section, we specify different parts of the system that require a separate monitoring and control mechanisms.

The first component of the layer is the environment information. This is a monitoring component which receives its information from two sources: the Context and the Activity Recognition. First of all, the component provides an important hint to the user about the view of the environment within the system, as generally it may be different from the actual state of the environment. Causes of this may be numerous: an erroneous reading of the sensor, a mistake of the Activity Recognition, missing information due to hidden changes that are not detected, etc. If the view within the system differs from the actual environment state, the decision may be incorrect or not optimal as well. Thus it is important for a user to be able to see the view within the system in order to be able to compare it with the actual state.

There is another important benefit of the environment information component: the increased user's awareness. Many studies show that just by increasing users' awareness about the amount of energy they consume at certain times and when using certain devices, it is possible to reduce the total energy consumption, because users are more likely to decrease their usage of heavy-consuming devices [Weiss and Guinard, 2010].

The second component of the User layer is the knowledge base information and update feedback module. The static information about devices, their configuration, possible actions, etc., is a great reference for a user about the capabilities of the system. Whether or not the component should provide the ability to update static information depends on the general architecture of the system, particularly on where the entry point of such information to the system is located. For example, if the system should be able to automatically detect and configure the device for work, it may be wiser to restrict the ability to tamper with the device parameters through the user layer. More often than not, incorrect detection may highlight deeper problems or bugs with device detection, which should be fixed, instead of just concealed by the manual correction.

The next component of the layer is the reasoning and decision making results module. This information helps to understand the origins of system's actions. It shows the reasons why a particular decision was made by the system. For example, if the system decides to perform a certain action, this component will highlight exactly which rules were activated. It is important to note that this includes information from all components of the Reasoning layer: the Decision Making, the Learning, and even the Activity Recognition. There is, however, no duplication of information with the environment information component, as the meaning of the information in these two cases is completely different. The environment information component must show the results of the activity recognition in order to show

how the system perceives the state of the environment. The reasoning results component, however, explains how and why the decision was made. Therefore it will show in details why the recognition algorithm classified the original information into exactly this activity, and not some other one. This knowledge helps the user to tweak the recognition algorithms if needed.

Finally, the last component of the User layer is the system monitor. Contrary to all previous components, instead of showing the information about the domain and the environment, this component shows the information about the system itself: health status of all components, their performance indicator, any detected status changes and/or errors, etc. This also includes detected errors in devices, which may require user's intervention in order to check if device is working properly or indeed needs to be changed or repaired.

3.2 Operational Flows

Intelligent building systems are reactive, i.e. their behaviour is a direct consequence of the information they get from outside. There are three general operational flows within the system, and every flow corresponds to a single information entrance point.

Of course, in our description it is assumed that all components are present in the system, which is not true for the general case, as many components are optional. If some component is missing, then every piece of information that should pass through the component is passed as it is (so we may assume that the transformation is the identity), and the component generates no new information.

3.2.1 Environment-generated

This flow is the most common one, as it starts with any registered change in the environment, and partially with every new sensor reading.

The sensor reading is generated in the Physical layer and is sent to the Common Gateway where it is converted to a uniform format. From the Gateway the transformed reading goes to two places: to the Knowledge Base for storage and further retrieval as historical data, and to the Context for immediate processing.

In the Context the reading is assessed and transformed from raw data reading into a higher-level information about the current state of the environment. It may be the case that the reading corresponds to no changes in a state, in which case, depending on the system, the flow may either stop here (if further components are only interested in changes), or go further as usual. Either a state or a raw reading

data (depending on the system) is also sent to the Diagnosis component, where it is checked for correctness.

The state is further sent from the Context to the Reasoning layer, starting with the Activity Recognition component, where recognition is performed to generate domain level knowledge. Then it is sent to the Decision Making component, where it is combined with all other available information and the system decides, whether a certain action should be performed.

In case there is a need for a certain action, the action is sent from the Reasoning layer to the Execution component, where it is transformed to a set of device-level commands. And finally, those commands are sent to the Common Gateway in order to be distributed among the corresponding devices. They are also sent to the Diagnosis component for further checks.

Of course, in parallel with the flow described above, the information is sent to the User layer to be displayed in a timely manner. As soon as the Knowledge Base receives the new state, it is reflected on the corresponding dashboard. The environment information dashboard shows the results of the Context and the Activity Recognition components, and the reasoning dashboard shows the decisions made.

3.2.2 User-generated

The alternative flow is the user-generated one. This flow starts when a user shows the desire to change something in the way the system currently operates. For example, a user may override a certain decision, or change the priority of rules, or manually change a state of the environment, in case it was recognized incorrectly.

The flow starts from one of the informational components of the User layer. When a user enters the change, it is processed and is sent to the respective component. For a manual change of the environment it would be either the Context, or the Activity Recognition, for a rule change it will be the Reasoning component, for a decision override it will be either the Reasoning, or the Execution component, etc. From there the flow goes further normally.

3.2.3 System-generated

The first type of the system-generated flows concerns the normal system operation, for example, when executing scheduled events. In such a flow, on earlier stages a plan or a schedule has been generated that required certain actions to be performed in the future. In such a case, the internal clock is set, and when the time comes, the action is automatically launched. The event usually starts from the Reasoning component, and goes further to the Execution normally.

Another type of system-generated flows concerns the re-learning and re-training mechanisms. Usually the Learning component is updated during the course of system's operation, in order to correspond to changing conditions and requirements. Updating after every state change may be too cumbersome, especially for computationally expensive machine learning methods. Thus, the re-learning happens either at some intervals of time, or when a certain condition is met (such as a threshold for amount of changes is achieved).

3.3 Challenges

For an intelligent environment that features a single room or a few rooms with no more than a couple of dozens of devices, the already described architecture will normally satisfy all demands of the architects and users combined. However, when a system becomes larger and grows to include several floors, a whole apartment or office building, or even several houses, new issues emerge that may render the intelligent environment almost non-operational until properly solved.

The scalability of the system is the first of such issues. First of all, a single server's CPU or storage power will be quickly outgrown. Currently, many efforts are spent in the area of database systems on development of distributed fault tolerant databases, such as Hadoop [White, 2012], MongoDB [Chodorow and Dirolf, 2010], Redis[1], Cassandra [Lakshman and Malik, 2009], and other noSQL databases. Such databases make a good foundation for extendable intelligent environments, as they already solve distribution, data replication, fault tolerance, and availability problems out of the box. However, not only the Knowledge Base needs proper scalability. The amount of sensor data grows with the number of devices as well, and at some point concurrency, queue processing speed and bandwidth issues may stop the system from further expansion. Thus it is also important to use proper solutions not only for data storage, but also for high-volume fast data processing. Such solutions as Twitter Storm[2] or RabbitMQ[3] [Videla and Williams, 2012] provide reliable ways for sending and processing large streams of data.

The Reasoning layer is the one that may suffer most from system's expansion. The reason is that most of the machine learning, search and reasoning algorithms within the layer may be computationally expensive with at least exponential solving time. While the parallelization and distribution on several servers may partially alleviate the problem, sometimes more fundamental changes to the algorithm will

[1] `http://redis.io/`
[2] `http://storm-project.net/`
[3] `http://www.rabbitmq.com/`

be required. One of possible changes is the usage of approximate algorithms (for example, the greedy or genetic algorithms) instead of exact ones for the optimization reasoning. Another possible change is the splitting of the system into several independent subsystems of smaller size and the application of the algorithms within subsystems. While with this approach some dependency between parts from different subsystems may be permanently lost, if the subsystems have only weak and not important dependencies between each other, this may be a big improvement in terms of system's reaction time with only minor consequences in terms of the optimality of reasoning results.

Another direct consequence of scaling the system into several distributed servers is the need to increase the fault tolerance level. If the system works only in one room and on one server, crashes and other unrecoverable faults are rare and restarting the system is an unpleasant, but fast procedure that has overall light consequences. However, when servers become numerous, the rate of errors and crashes increases as well. The system should be designed in such a way that any single error will cause only a minor outage. So, for example, the system should be fully operational on fifth floor of the building even if the server that manages the second floor crashes.

This may be achieved through the addition of special system-level components, i.e. components that manage the system itself. Monitoring and configuration component may keep track of all running instances of components and their servers, check their health status through heartbeats, and keep track of their configuration.

In case a component dies, the configuration component will automatically restart it either on the same server, or on a different one, and reconfigure other components so that now they contact a new instance. The configuration component may also perform load balancing and other utility tasks. As with databases and data streams, there are solutions that may come handy for such component implementation, such as Apache Zookeeper[4] or Doozer[5].

3.4 Case Studies

In this section we showcase several prominent smart environment projects as case studies and discuss how their architecture maps to the general pattern just described. These projects are chosen due to several factors. First of all, their focus is on the creation of a fully featured general intelligent building which influences all aspects of building's operations, as opposed to specifically targeted projects, such as those that aim to create a smart lighting system, or those that only target

[4] `http://zookeeper.apache.org/`
[5] `http://github.com/ha/doozer`

efficient system's infrastructure. Secondly, all chosen projects have constructed, implemented and tested an actual real environment, thus the architectures of these projects have proved their feasibility and validity. And finally, they mostly feature clear distinction of architecture modules, as opposed to several smaller projects, where some modules can be seamlessly combined, or removed altogether, due to their reduced functionality.

Even though the presented pattern is the most commonly used one for smart buildings, sometimes specific requirements may induce other constraints on the project and its architecture. For example, an emerging view of smart home architectures is viewing smart building environments as multi-agent. [Cook, 2009] defines four different directions in multi-agent research of smart environments: (a) multi-intelligent software agents, (b) tracking multiple residents, (c) profiling multiple residents, and (d) multi-agent negotiations. The first direction usually assumes viewing every module of the system as a separate agent, with communication protocols guiding interactions between them. Surprisingly, such a view of multi-agent architecture can be very well combined with the pattern presented here. In fact, in the same work Cook uses the MavHome project, which is one of our case studies as well, to describe how the agents can be organized in a hierarchical layered configuration. Other research directions view as agents either different people (in which case the smart system itself remains unified, but has to incorporate additional intelligence for distinguishing people), or different devices. In the latter case, especially if devices are highly mobile and autonomous, thus may be viewed as a complete system by themselves, the proposed pattern may be inapplicable or sub-optimal, and other agent based architectures may be explored, for example as described in [Spanoudakis and Moraitis, 2006].

In the project case studies below, to avoid confusion, we refer to the layers of the architecture pattern, which is described in this chapter, as "the pattern architecture", and to the layers of respective projects as "the project architecture".

3.4.1 MavHome

The Managing An Intelligent Versatile Home (MavHome) project was one of the first scientific projects to create a functioning smart environment [Das et al., 2002]. The home system in the project acted as a rational agent, whose goal was to maximize comfort of its users and minimize costs of operation. The project used learning and prediction techniques heavily to predict mobility patterns of the inhabitants and adapt to them in a timely manner.

The architecture of the MavHome project is described in details in [Youngblood et al., 2004]. It is divided into four main layers: Physical, Communication,

Information, and Decision.

The Physical layer of the MavHome exactly maps to the Physical layer as described in the pattern: it contains physical components, i.e. physical devices, and hardware interfaces between devices and higher-level components, reminiscent of the Common Gateway of the pattern.

The Communication layer contains many utility components that help to make the system operational, such as device drivers, operating system, proxies, and middleware. When comparing to the pattern, the Execution component is a part of this layer of the MavHome. As we mentioned at the beginning of the Section 3.1, we avoid to include any implementation details and support components into the pattern, as they are very specific to every system, however they may very well be present in the high-level architecture overviews of particular projects. The Middleware sub-layer of the Communication layer of the MavHome project contains such components, as Bootstrap for component mobility, ZeroConf for naming and discovery, and CORBA as a point to point publish/subscribe system. One thing to note is that all device and hardware related utility software, such as drivers, operating system, or proxies, may also be conceptually viewed as a part of the Physical layer of the pattern.

The Information layer of the MavHome contains aggregator, prediction, data mining and database services. It combines into a single layer parts of both the Ubiquitous and the Reasoning layers of the pattern. Namely, the Knowledge Base and the Context from the Ubiquitous layer, and the Learning and the Activity Recognition from the Reasoning layer.

Finally, the Decision layer of the MavHome project corresponds to the Decision Making component of the pattern, and contains component applications such as Decision Maker, Learning, and Policy.

In the MavHome architecture there is no specific component or layer, responsible for interfacing with the user, even though such interfaces (including mobile interface on PDA) actually exist. In case of their inclusion into the architecture picture, they may constitute the next layer, similar to the User layer of the pattern.

3.4.2 SmartLab

The SmartLab is another project that has created a functioning smart environment [López-de Ipiña et al., 2008]. The uniqueness of the project lies in the fact that the project itself features a hardware and middleware parts of the environment (the Physical and the Ubiquitous layers in the pattern), with common interfaces for other projects to use and to create their own reasoning on top of it (the Reasoning layer of the pattern).The SmartLab environment was already used as a base

for several other research projects, including Assistive Display[6], ubiClassRoom[7], Eldercare[8].

The architecture of the project is described in details in [López-de Ipiña et al., 2008]. The architecture contains four main layers: Sensing and Actuation (devices), Service Abstraction (embedded platform), Semantic Context & Service Management (SmartLab server), and Programming, Management & Interaction (applications).

The Sensing and Actuation layer contains all devices within the environment. They include EIB/KNX bus for lightning, HVAC, presence, temperature and motors on doors and windows, VoIP and VideoIP, Indoor Location System, etc. The next layer is the Service Abstraction layer, which transforms functionality of the devices from the first layer into software services. Together these two layers represent the Physical layer of the pattern, with the second layer representing the Common Gateway.

The Semantic Context & Service Management layer contains the Service Manager, which monitors the environment for activation and deactivation of devices thus for availability of services, the Semantic Context Manager, which stores knowledge about devices and rules in the common ontology, and the Web Gateway Module, which produces interfaces for third-party programs wishing to interact with the environment. This layer corresponds to the Ubiquitous layer of the pattern, with the Service Manager behaving as the Context component, the Semantic Context Manager behaving as static storage of the Knowledge Base component, and the Web Gateway behaving as the Execution component.

Finally, the Programming, Management and Interaction layer provides web-based interface for users of the SmartLab laboratory. The Environment Controller allows a user to manually operate the environment through a set of widgets, while the Context Manager Front-End offers a web interface for management of devices configuration, ontology, rule behavior, and tracking the system log and statistics. As can be seen, the layer closely resembles the User layer of the pattern.

Note that there is no layer similar to the Reasoning layer. As we already mentioned, the project provides capabilities for external programs to use the environment and middleware while applying their own reasoning. Thus such external programs will represent the Reasoning layer, when attached. Instead, the Semantic Context & Service Management layer provides all interfaces needed for external programs.

[6] `http://www.smartlab.deusto.es/assistive_display/`
[7] `http://www.smartlab.deusto.es/ubiClassRoom/`
[8] `http://www.smartlab.deusto.es/eldercare/`

3.4.3 Smart Homes for All

Smart Homes for All (SM4All) was a European-wide research project that created a smart apartment in Rome, Italy [Aiello et al., 2011]. The project featured several innovative ideas within smart environments, including usage of brain computer interfaces for issuing the commands, using planning techniques for finding a set of actions for complex commands, and sophisticated execution mechanisms to avoid concurrency issues when executing the commands.

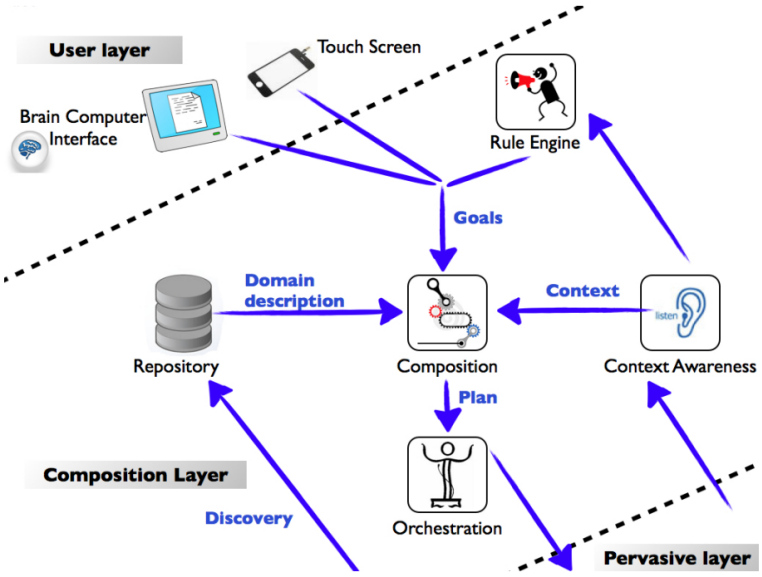


Figure 3.2: SM4All Architecture

The architecture of the project as described in [Aiello et al., 2011] can be seen in Figure 3.2.

There are three main layers. The Pervasive layer contains all devices and gives the possibility for devices to be added or removed dynamically through the usage of the common Universal Plug and Play (UPnP) protocol. The layer has the direct correspondence to the Physical layer of the pattern.

The Composition layer contains five major components. The Repository represents the Knowledge Base component of the pattern, and contains a database, which includes registry of current devices and their abstract types, description of available services, and information about the layout of a house. The Context Awareness collects sensed data and represents the logical image of the environment, thus being

the Context component of the pattern. Though there is no specific Activity Recognition component from the pattern included in the SM4All architecture, some parts of it are also included in the Context Awareness.

The Orchestration component controls the execution, i.e. it invokes the physical services and receives feedback about the status of invocations. As such it corresponds to the Execution component of the pattern. The Rule Engine component contains rules of the environment behavior and constantly checks, based on information from the Context Awareness component, whether those rules are satisfied; if so, it invokes the Composition component, which applies AI planning techniques to create a set of actions which are sent to the Orchestration. The Rule Engine and the Composition combined constitute the Decision Making component of the pattern.

The User layer provides access to the home system to its users. Users may issue direct commands either through the touch interface or through the Brain Computer Interface. The User layer corresponds to the Reasoning results component of the User layer of the pattern.

3.4.4 GreenerBuildings

The GreenerBuildings project is the project that is dedicated to creation of smart offices in a green and energy-efficient way, while maintaining the high level of occupants' comfort [Degeler et al., 2013]. Occupants' behavior and activities are the key for adaptation to maximize the comfort, while choosing the most energy efficient state. The living lab setting is constructed on the premises of Eindhoven University of Technology, the Netherlands. The project focuses on the creation of a scalable, distributed, and fault tolerant solution. The architecture of the project is shown in Figure 1.1.

The Physical layer contains all devices connected to the Sensors and Actuators Gateway, which sends the values further into the system. As such the Physical layer of the project resembles closely the Physical layer of the pattern. Note that the project layer contains one more component: the Interconnection with Smart Grid. Since the project puts many efforts in energy saving, the Smart Grid component provides the energy consumption and energy costs information. It also provides prices of energy from different energy providers, so that it is possible to choose the best price and the best time of task executions when the prices are the cheapest. The Interconnection with Smart Grid is a component, specific to the implementation of the GreenerBuildings, so there is no corresponding component in the pattern. However, since it provides information, as other devices do, it can be viewed as a part of the usual Physical layer subsystem.

The Ubiquitous layer contains three main components: The Context, the Repository and the Orchestration, each having more subsystems within it. The Repository contains information about device types, device instances, and saves historical data for further retrieval. It corresponds to the Knowledge Base component of the pattern. The Context component collects information from sensors and transforms it to offer a consistent view of the environment. It also performs activity recognition and as such it combines two components of the pattern: the Context and the Activity Recognition. The Orchestration performs execution of commands and also diagnoses errors on the Physical layer. Therefore it combines the Execution and the Diagnosis components of the pattern.

The Composition layer contains two main components: the Control and the Composition component. The Composition component contains the reasoning of the system. The Rule Maintenance system within the component uses constraint satisfaction techniques to constantly check all rules that users have added to the system, and finds the state of the environment which satisfies all the rules. Planning component creates a set of actions to be executed by the Orchestration, and the CFD is the special system for optimal handling of the heating mechanisms and air quality within the rooms. Thus the Composition component is the Decision Making component of the pattern.

The Control component is the main system interface to a user. It shows system's parameters, and allows a user to issue direct commands or overrule decisions of the system. It also collects information about the users' satisfaction levels. As such it partially corresponds to the User layer of the pattern.

Chapter 4

Dynamic Constraint Reasoning in Smart Environments

In any smart environment, the autonomy and reasoning power should be counterbalanced by the ability of users to fully understand the reasons of the system's automated operations and their ability to fully control the system's decisions, adapting them to their goals and desires at any moment. Therefore, flexible and adaptable reasoning mechanisms are essential for environment automation.

Our approach, implemented in the GreenerBuildings project, is to specify scenarios of the building's operations via sets of logical rules. The predefined sets of rules for standard behavior may always be modified or fully overridden on global or local levels by facility managers or particular users.

The rules combine context information about the environment with the desired behavior of actuators, and must at all times be satisfied whenever it is possible, or be able to communicate failure to relevant users when impossible.

This behavior can be modelled as a constraint satisfaction problem (CSP). In particular, the model falls into the dynamic constraint satisfaction problems (DCSP) [Verfaillie and Schiex, 1994] category, due to the necessity to solve the satisfiability problem over and over again, every time with small changes (due to changing context) from the previous task. If costs are involved, for example the desire to find the most energy efficient way to satisfy the current set of rules, the usual CSP task may need to be solved as an optimization CSP.

In this chapter, we present the dynamic constraint satisfaction solution for the GreenerBuildings project. First of all, we explain why the straightforward encoding of the problem to the (D)CSP task brings suboptimal efficiency, and how the specific structure of the smart environments domain can be exploited in order to make CSP models smaller and decrease amount of computations required to solve every subsequent CSP task. In particular, the existence of context variables (information from sensors) and controllable actuators, and the uneven dependency of variables are exploited. By uneven dependency we mean the existence of highly dependent subsets of variables (for example, devices that are part of a common

area within a single room) with many interconnecting rules, which have very loose or no dependency on another subset of variables (e.g. devices from a different room). The main contribution involves the formulation of the dependency graph data structure, which makes it possible to split CSP into dynamically independent subtasks, and to find only the affected parts of the problem every time a new event arrives to the system, which severely reduces the size and complexity of the CSP to be solved at every subsequent step. We also present a specific method to transform rules into a form which makes the dependency graph possible.

4.1 Rule Satisfaction in Smart Environments

The GreenerBuildings project aims to increase the overall users' comfort by adapting to their needs. Usually there are several ways to satisfy them. Therefore, the additional goal of the project is to assure the minimum energy consumption of the building, without sacrificing user comfort. The reasoning is handled by the Rule Maintenance Engine (RME) component. Via the web interface the users are able to access the current rules, modify them, add new rules, or delete obsolete ones. The information about the current state of the environment comes from the Context component as new sensor readings events. Informally, the RME goal can be defined as follows:

Given a set of user-defined rules of the building's behavior, and information about the current environment state, the Rule Maintenance Engine must ensure that:

1. *The rules are satisfied and adhered to. If there are some rules which cannot be satisfied at a given moment, the user must be presented with sufficient information to identify the cause.*
2. *While satisfying all rules, the energy consumption of the building should be minimal.*
3. *Decisions should be made in real-time and be scalable with respect to the environment size.*

In general, rules are entered to the system by its users. However, there are certain "ready-made" presets of rules that users may use. The system gives the ability to modify sets of rules or switch between different sets.

Rules describe the expected and desired behavior of the smart building. In general there are two different types of rules. The RME system itself handles those types equivalently, but for the users of the system they represent a difference between what is *necessary* and what is *desirable*.

The first type represents a *dependency between variables*. For example, a rule $desk1.monitor = active \Rightarrow desk1.pc = on$ tells the system that it is not possible to have a monitor in an active state if the PC to which the monitor is connected is off. The second example is $\neg(room1.blinds1 = down \wedge room1.window1 = open)$, which represents a physical constraint that blinds can only be put into down position if the window is closed.

The rules of the second type are in essence *user preferences*. They describe the desired behavior of the system. For example, a rule $room1.presence > 0 \Rightarrow room1.ceilinglamp = on \vee room1.desklamp = on$ represents a desire to have a light on in the room, if there are people inside.

The rules are defined as formulas in a predicate logic over finite domains. Every atomic predicate represents a certain condition over a variable, and should result in $true$ or $false$. There are several available operations in predicates. The equality represents that a variable should be equal to a given value for a predicate to be true. For example: $room313.dimmer1 = 0$. Opposite to it, the inequation is used to forbid a variable to be equal to a certain value, e.g. $room313.dimmer1 \neq 0$. It is also possible to use a set of values instead of a single value in both cases, e.g. $room313.dimmer1 \in \{0; 10; 20\}$ or $room313.dimmer1 \notin \{0; 10; 20\}$. These operations are available for all types of variables. For ranged variables, i.e. integer or real ones, it is also possible to use inequalities, i.e. greater (or equal) / less (or equal) than. For example: $room313.dimmer1 > 50$; $room313.dimmer2 \leq 200$. To summarize, the rule with only a single atomic predicate is represented as:

$$P ::= (v_i = d) \mid (v_i \neq d) \mid (v_i \in \{d_i\}) \mid (v_i \notin \{d_i\})$$
$$P ::= (v_i < d) \mid (v_i > d) \mid (v_i \leq d) \mid (v_i \geq d),\ v_i \in \mathbb{R}$$

Of course, atomic predicates can be combined together to form logical formulas of any additional complexity, using the standard logical operators:

$$R ::= P \mid \neg R \mid R \wedge R \mid R \vee R \mid R \Rightarrow R \mid R \Leftrightarrow R$$

4.2 Environment Definition as CSP

The environment $\langle V, D \rangle$ is defined by a set of context variables $V = S \bigcup A$; $S \bigcap A = \emptyset$, where $S = \{s_1, s_2, ..., s_n\}$ is a set of uncontrollable variables, and $A = \{a_1, a_2, ..., a_m\}$ is a set of controllable variables. Uncontrollable variables S represent sensors, they provide information about the environment, and cannot be directly influenced by the system. They do not necessarily represent a physical sensor. A variable can represent a combined value of several sensors, or a result of a certain activity recognition task.

On the other hand, controllable variables A can be seen as actuators that can act in the environment in an automated way, i.e. by receiving appropriate commands from the system. We assume that it is possible to change the state of every actuator independently from other actuators, and that it is possible to transform an actuator from any state of its domain to any other state of its domain.

Every variable $v \in V$ varies over a finite *states domain* $d(v)$ with size k_v, which can be either a range of integer or real values, a boolean, or a set $d(v) = \{d_{v1}, d_{v2}, ..., d_{vk_v}\}$. Each variable v has a *cost function* $c_v(d_i)$, associated with its state domain $d(v)$ that shows the cost of keeping the variable in this state. For the GreenerBuildings project the cost is associated with the energy consumption of corresponding devices.

The original set of rules R_o contains a set of logical formulas over variables in V. Every rule $r \in R_o$ can be represented as a constraint to the classical CSP model, which corresponds to a subset of variables $V_r = \{v_{r1}, v_{r2}, \ldots\}$, and represents a subset X_r of a Cartesian product over their respective domain values $d(v_{r1}) \times d(v_{r2}) \times \ldots$, which specifies the sets of values of those variables that are compatible with each other. This subset can be trivially constructed by constructing the full truth table for a set of variables V_r, and retaining only those values from a table, for which the rule evaluates to *true*.

It is possible to use the original set of rules R_o as a set of constraints to the CSP task, though we also need to add the knowledge about the current sensor values to the problem definition, since we know their values from the context environment information, and we cannot influence them directly. For every sensor $s \in S$, if its current value is d_s, one more rule $s = d_s$ is added to restrict the sensor. In this case, the natural constraint satisfaction problem for the smart environment will be defined as follows:

Find a valuation for a set of variables $V = S \bigcup A$ which satisfies all constraints $C = R_o \bigcup R_s$, where R_o is the original set of predefined rules, and R_s is a set of sensor constraints for every sensor: $\forall s \in S : s = d_s$, where d_s is the current sensor value of the sensor s, obtained from the context information. We will refer to this CSP definition as $CSP(V, R_o \bigcup R_s)$.

Such CSP representation, however, is very inefficient in practice with respect to the amount of required computations. The reasons for this are the following:

- In order to keep the solution valid and up to date, the CSP task should be solved for every new sensor change event. For the smart buildings with hundreds of sensors several of such events arrive every second. Solving the CSP for the full environment is a computationally heavy task, and doing it for every new sensor change event can represent a big strain on resources.

Such solution has a very low scalability potential.

- Every sensor change affects only a small part of the environment, therefore solving from scratch every time produces a large amount of duplicate work. Using dynamic constraint satisfaction techniques is more computationally efficient.
- In practice, many rules (constraints) for intelligent environments are only applicable for a particular situation, which may occur only a small percentage of the time. For most of the time the constraints will not be applicable, however they will still need to be added as a part of the CSP over and over again.

The classic definition of Dynamic Constraint Satisfaction Problem [Dechter and Dechter, 1988; Bessiere, 1991] defines it as a set of successive CSPs, where every next CSP is created from the previous one by adding or removing a variable or a constraint. Though in our case most of the changes to the environment do not involve direct addition or removal neither of a variable, nor of a constraint, we can still represent a problem in such a way, by representing a change of a sensor value $s \in S$ from d^s_{old} to d^s_{new} as removal of a constraint $s = d^s_{old}$ and addition of a constraint $s = d^s_{new}$.

In the classic definition the domain remains of the same size. On the other hand, our solution for dynamic constraint satisfaction of smart environments allows to make the problem domain smaller for every subsequent CSP, by reusing dynamically independent parts of the previous problem.

4.3 Rule Transformations

Users may enter rules in any form they like, but to make the automated processing easier, the rules are transformed into a special uniform way. Transformations are done only once at the time of addition of a new rule by users (or after a rule has been modified), and should ensure that the least amount of processing is kept for the real-time system's operation. There are two reasons for transformations.

First of all, we split the rule into as many independent sub-rules as possible. For example, a rule $chair = occupied \Rightarrow pc = on \wedge lamp = on$ should be split into two different rules: $chair = occupied \Rightarrow pc = on$ and $chair = occupied \Rightarrow lamp = on$. This will not change the overall rule satisfaction logic, as all the rules should be satisfied, however, such splitting ensures that we do not register a false dependency between two variables "*pc*" and "*lamp*", as it can be seen that, at least if using only this rule, they may be satisfied or not satisfied independently.

The second reason is that at the end we want all resulting rules to have a form $F_s(S) \Rightarrow F_a(A)$, i.e. some function of sensors implies a function of actuators. The benefits we achieve with this are twofold. First of all, the sensors S cannot be influenced by the system, thus they represent the situation that is given to us. There is no possibility to directly influence the antecedent of the equation $F_s(S)$; with the given context in the current situation it is either satisfied or not. If it is not satisfied, or let us rather say "the situation described in $F_s(S)$ does not occur", then we do not need to do anything about the consequent of the equation, the $F_a(A)$, which contains actuators, as the full equation is already satisfied. The rule is then in the "inactive" state, i.e. it is possible to skip it in the constraint satisfaction problem, which can help us to severely reduce the search space and decrease dependencies. If, on the other hand, the $F_s(S)$ is met, i.e. results to *true*, then we must ensure that the consequent, which contains actuators $F_a(A)$, is satisfied. Thus the second benefit. Since we can only control actuators, only actuator variables are meaningful for the CSP search space. When we use such a form, we can only put $F_a(A)$ part of the formula to the CSP description, and only when we actually need it to be satisfied.

Finally, to ensure the fastest processing the functions $F_s(S)$ and $F_a(A)$ are transformed into the form $\bigwedge_s(P(S)) => \bigvee_a(P(A))$. Here $P(S)$ and $P(A)$ are atomic predicates with respective variables. The form $\bigwedge_s(P(S))$ ensures that with every new sensor reading $s = d_s$ it is possible to recheck only a single atomic predicate $P(s)$. The form $\bigvee_a(P(A))$ is the easiest for CSP solvers to work with.

It is always possible to transform any human-defined rule into such a form. The actual transformation is done in the following steps. First of all, the original rule is transformed into the CNF form. Every conjuncted clause (the disjunction) in the CNF form is connected by $\wedge$-clause and, since all rules must be satisfied, may be regarded individually. Therefore every such a clause will represent a single separate rule in the resulting set, so often an original rule will result in several final rules. Every resulting rule is a disjunction of atomic predicates (possibly negated). On the second step it is transformed into an implication by taking those atomic predicates that contain only sensors, and putting them (in negated form) into the antecedent of the implication. The next step is not necessary, and is done only for convenience, in order to unify further representation and processing of transformed rules: negation is removed from all negated atomic predicates by flipping the operation. For example, the $\neg(room1.dimmer1 > 100)$ becomes $room1.dimmer1 \leq 100$, and $\neg(desk1.pc = on)$ becomes $desk1.pc \neq on$.

For example, let us assume we have a rule that requires to have light in the room if there are people inside. Light can be achieved either by turning on the

lamp, or by opening the blinds, but only in case there is enough light outside:

$$room1.presence > 0 \Rightarrow \begin{array}{l} room1.lamp = on \vee \\ outsidelux > 1000 \wedge room1.blinds = open \end{array} \tag{4.1}$$

Sensors here are $room1.presence$ and $outsidelux$. So, by putting it into CNF, splitting it into two distinct rules, putting the sensors to the antecedent, and removing the negation from atomic predicates we obtain the following two rules:

$$room1.presence > 0 \Rightarrow room1.lamp = on \vee room1.blinds = open \tag{4.2}$$

$$room1.presence > 0 \wedge outsidelux \leq 1000 \Rightarrow room1.lamp = on \tag{4.3}$$

If someone is present in the room, the first rule will be "active" and the system will need to either turn on the lamp or open the blinds. In practice, since optimization CSP is used, if both choices are not restricted the system will choose to open the blinds as more energy efficient choice. But if the outside light level is sufficiently small, the second rule will also become active, which means the only choice left will be to turn on the lamp, as it will satisfy both rules.

4.4 Dynamic Dependency Graph

The environment size for pervasive smart buildings may become considerably large, easily reaching hundreds of variables. One of the goals of the RME component is to ensure that such environments can be handled in real-time, thus rechecking all variables after every event registered by one of the sensors is definitely a non-practical solution.

It is better to recheck only parts of the environment, which are actually affected by a change. This is, however, not always a straightforward task. Dependencies are introduced via rules, but it is not enough to recheck all the rules that contain the changed sensor to find a new optimal state of the environment, as easily shown by the earlier example that requires either the lamp to be on or the blinds to be open during the day, if people are inside the room. Let us assume that rules (4.2) and (4.3) compose our ruleset. When it is still dark someone enters the room, so sensor values are $room1.presence > 0$ and $outsidelux = 500$. The only way to satisfy both rules is to turn on the lamp, so the system does it, while keeping the blinds shut. Now the outside light gradually increases, and at some point becomes bigger than our threshold: $outsidelux = 1100$. At this moment only the sensor

from the rule (4.3) is changed, and the rule is not active anymore. But because of this change the first rule (4.2) can now be satisfied in a different way, by opening the blinds, which is more energy efficient, so it also needs to be rechecked.

Another option is to transitively consider all variables affected, if they are a part of the affected rules. However, this will largely overestimate the amount of rules and variables to be rechecked. For example, assume we have a rule $(desk1.chair = occupied \wedge desk1.paperwork = true) \Rightarrow desk1.lamp = on$, and the chair becomes occupied, while the paperwork does not change and remains false. In this case the total antecedent of the rule has not changed, it is still not satisfied, thus we should not even trigger the rechecking of the lamp and all other variables, which may be dependent on it.

The dynamic dependency mechanism, which is realised via the use of the Dependency Graph is specifically the mechanism designed to keep track of the actual dependencies between the variables, based on the context information, and only invoke re-optimization tasks for the smallest subsets of the variables which are actually affected.

As shown in Section 4.3, after performing rule transformations, we obtain an internal set of rules R, where every rule $r \in R$ is in a form $F_s(S) \Rightarrow F_a(A)$, specifically $\bigwedge_{s \in S}(P(s)) \Rightarrow \bigvee_{a \in A}(P(a))$.

First of all, sensor variables should be removed from the CSP model. At every moment in time sensor variables have a particular valuation, based on the context environment information, and represented by a set of rules R_s: $s = d_s$, $\forall s \in S$. Therefore, while the sensor values influence the valuation of actuators, the sensors themselves are not *decision variables*, as only a single value is applicable to them, and we know this value in advance.

The rule form $F_s(S) \Rightarrow F_a(A)$ helps to construct an equivalent CSP model that does not contain sensor variables. For this, we define an *active* property of rules:

Definition 1 (Active/inactive rule)**.** A rule $r = (F_s^r(S) \Rightarrow F_a^r(A))$ is *active* in the current state of the environment, i.e. with a given valuation of sensors R_s, if the antecedent part of the rule $F_a^r(A)$ valuates to *true*, and *inactive* otherwise.

Let $R^* \subseteq R$ represent an active subset of rules R.

If the rule is inactive, it poses no constraint for the actuator values, as the full rule is already satisfied regardless of them. So the rule may be removed from the CSP model at this moment in time. The activeness of a rule changes with time and different sensor values.

Using the notion of rule activeness, we change the previous CSP definition:

$$CSP(V, R_o \bigcup R_s) \equiv CSP(A, F_A^R),$$

$$\text{where } F_A^R = \{F_a^r(A)\},\ \forall F_a^r(A) \text{ of } r \in R^*$$

Not only such definition removes all sensor variables from every consecutive CSP task, but also many original rules are removed, leaving only those that are actually relevant to the current situation and state of the environment. Given the nature of smart environment rules, it is usually a small subset of the original rules at any moment in time.

The next step in transforming the task definition is to find sets of dependent variables. For this, we formally define dependency of variables and rules:

Let $X(V_x)$ represent the set of *full Cartesian product of values* for a variable set V_x: $d(v_{x1}) \times d(v_{x2}) \times \ldots$.

Let $r(x)$ for $r \in R$ and $x \in X(A)$ identify the result of evaluation (*true* or $false$) of the consequent actuator part $F_a(A)$ of a rule r with actuator values in valuation x.

If $B_r = \{a_{r1}, a_{r2}, \ldots\}$ is a subset of actuators $B_r \subseteq A$, then let NB_r be a complement subset: $NB_r = A \backslash B_r$.

Definition 2 (Dependency)**.** The rule $r \in R$ is said to introduce a *dependency* over a subset of actuator variables $B_r = \{a_{r1}, a_{r2}, \ldots\}$ (or, alternatively, a rule r depends on variables $a_{r1}, a_{r2}, \ldots$), iff:

1. $\forall x \in X(NB_r) : \exists w_1, w_2 \in X(B_r)$ s.t.: $r(x \times w_1) \neq r(x \times w_2)$
2. $\nexists a_{nb} \in NB_r$ s.t. $\exists d_1, d_2 \in d(a_{nb})$, $\forall x \in X(NB_r \backslash a_{nb})$, $\forall w \in X(B_r) :\ r(d_1 \times w \times x) \neq r(d_2 \times w \times x)$
3. $\nexists a_b \in B_r$ s.t. $\forall d_1, d_2 \in d(a_b)$, $\forall w \in X(B_r \backslash a_b)$, $\forall x \in X(NB_r)$: $r(d_1 \times w \times x) = r(d_2 \times w \times x)$

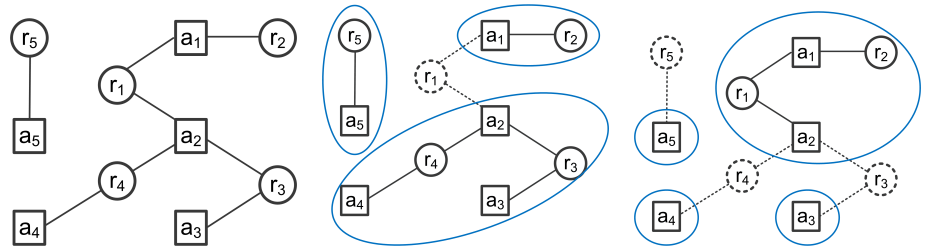
The first part ensures that the result of a rule evaluation will indeed change with different valuations of variables in B_r.

The second part ensures that the set B_r is *complete*, i.e. there is no variable outside of this set, s.t. changing a value of this variable will still result in a change of a rule evaluation result.

The third part ensures that the set B_r is *minimal*, i.e. there is no variable in this set, which does not influence the evaluation result irrespectively of its value.

We use the dependency relation to find subsets of dependent variables and rules. To do it, we introduce a *dependency graph*:

Definition 3 (Dependency graph)**.** The dependency graph for a set of actuators A and a ruleset R is a bipartite graph $G = \langle A, R, E \rangle$, where A and R are two sets



(a) All rules are active (b) r_1 is inactive (c) r_3–r_5 are inactive

Figure 4.1: Dependency graphs

of vertices, and $E \subseteq A \times R$ is a set of edges, $(a, r) \in E$ iff the consequent part $F_a(A)$ of the rule r depends on a.

Figure 4.1a shows a dependency graph example. Two disconnected subgraphs in the figure represent a *static independency*, i.e. there is no rule that may potentially make the variables from different connected subgraphs dependent on each other. Every rule and every variable are a part of only a single subgraph, and it is clear (see Lemma 1 for proof) that instead of having a single big CSP with all variables and rules combined, it is possible to "divide and conquer" by creating several smaller CSPs for every independent subgraph.

But most of the division benefits are gained not from static, but from *dynamic independency*, which exists when there are no *active* rules that make the variables mutually dependent. This dependency changes over time and with different sensor values, so two variables may be dynamically dependent at one moment, and independent at the next one.

Definition 4 (Active subgraph)**.** At a certain moment in time, an *active subgraph* of the dependency graph G is a connected subgraph of G that consists only of active vertices.

Examples of active subgraphs are shown in Figures 4.1b and 4.1c. By using this notion, we show that our solution to the DCSP of smart environments is globally optimal even with partial environment rechecking. First, we prove a lemma that smaller-sized CSPs for connected active subgraphs can be solved independently. Then, we prove the main theorem, stating that at every subsequent step it is possible to only recheck those subgraphs that changed their structure.

Lemma 1. For any set of solutions $x_i \in X(A_i)$ for all connected active subgraphs $G_i \subset G$, $G_i = \langle A_i, R_i^*, E \rangle$ s.t. $\bigcup_i(G_i) = G$, $\bigcup_i(A_i) = A$, $\bigcup_i(R_i) = R^*$, their

combination $x = \bigcup_i (x_i)$ is a solution of the full $CSP(A, F_A^R)$, $\forall r \in R^*$. And vice versa, if $x \in X(A)$ is a solution to the full CSP, when split into subsets of variables per active subgraph, these values will be a solution to smaller CSPs for connected active subgraphs: $CSP(A, F_A^R) \equiv \bigcup_i CSP(A_i, F_{A_i}^{R_i})$

Proof. We split the proof into two parts. First we prove that if $x \in X(A)$ is a solution to $CSP(A, F_A^R)$, then all x_i which are parts of the x that contain variables from active subgraphs G_i, are solutions to respective $CSP(A_i, F_{A_i}^{R_i})$. Then we prove that if $\forall i$: x_i is a solution to the $CSP(A_i, F_{A_i}^{R_i})$ of the subgraph G_i, then $x = \bigcup_i (x_i)$ is a solution to the full $CSP(A, F_A^R)$.

1. Assume $x \in X(A)$ is a solution to $CSP(A, F_A^R)$. Then x must also be a solution for a $CSP(A, F_A^{R_i})$, $\forall i$, since these CSPs contain the same set of variables A, but only a subset of original constraints $R_i \subseteq R^*$, therefore are less restrictive. From Definition 2 and Definition 4 it follows that the satisfaction of constraints R_i from active subgraph G_i depends only on variables from subset A_i, irrespectively of values of variables $A \backslash A_i$, thus the rules R_i are satisfied by valuation of $x_i \in X(A_i)$, $x_i \subseteq x$, therefore the smaller $CSP(A_i, F_{A_i}^{R_i})$ must also be satisfied $\forall i$.

2. Assume that $\forall i$: $x_i \in X(A_i)$ is a solution to $CSP(A_i, F_{A_i}^{R_i})$. If we add new variables $A \backslash A_i$ (to the total set of A) for every such CSP to obtain $CSP(A, F_A^{R_i})$, it will be satisfied for any valuation of new variables, since by Definitions 2 and 4 no constraint out of R_i changes its satisfaction status no matter the values of $a \in A \backslash A_i$. Therefore we can use valuation $x = x_1 \times x_2 \times \ldots$ to satisfy all $CSP(A, F_A^{R_i})$. So, the valuation x satisfies all rules in every set R_i. Therefore it must satisfy all rules in a combined set $R^* = \bigcup_i R_i$, $\forall i$, which means the valuation x must be a solution to the $CSP(A, F_A^R)$. □

Since every cost function only depends on a single variable, if x is optimal for $CSP(A, F_A^R)$, all $x_i \subseteq x$ must also be optimal for the respective smaller CSPs. Otherwise, if a valuation x_i' is better for $CSP(A_i, F_{A_i}^{R_i})$, following the chain of reasoning from part 2 of the proof, we arrive to conclusion that valuation $x' = x \backslash x_i \cup x_i'$ must also be a solution to $CSP(A, F_A^R)$, and it must be better than x, which contradicts the premise. And vice versa, if all independent subsets x_i are optimal, the full set x must also be optimal.

During the operation of the smart environment system, new sensor readings arrive as events. The change in a sensor value may potentially cause some rules to change their activeness status. The check takes constant time for every rule, as the form $\bigwedge_{s \in S}(P(s)) \Rightarrow \bigvee_{a \in A}(P(a))$ ensures that only a single atomic predicate $P(s)$ for a sensor s may change, and needs rechecking. Only the change in activeness status affects the actuators, and only a small percentage of new sensor

readings actually change the activeness of a rule, which saves the system from many unnecessary CSP solution invocations.

The change of activeness status changes the structure of active subgraphs around the rule. Either a single subgraph has one more (one less) constraint, or two or more subgraphs may join into one (one subgraph split into two or more).

We now prove the main theorem for DCSP in smart environments:

Theorem 1. For every event in the system, only active subgraphs that changed their structure must be rechecked for the whole valuation of actuators to remain satisfied and optimal.

Proof. Let x^t be the optimal solution found for the $CSP(A, F_A^{R^t})$ at time t, with active rules R^t. Let G^t represent a set of active subgraphs G_i^t at time t. Let x^{t+1}, $CSP(A, F_A^{R^{t+1}})$, R^{t+1}, G^{t+1} represent same notions for the time $t+1$.

We split x^t to a set of valuations $\{x_i^t\}$ that correspond to active subgraphs G_i^t. As proven in Lemma 1, every x_i^t is a solution to a corresponding $CSP(A_i, F_{A_i}^{R^t})$.

Let a sensor change at time $t+1$ make ruleset R_-^{t+1} inactive and ruleset R_+^{t+1} active. The total active ruleset at time $t+1$ is thus $R^{t+1} = R^t \backslash R_-^{t+1} \cup R_+^{t+1}$, and the rules $R_{const} = R^t \backslash R_-^{t+1}$ are active at both times t and $t+1$.

Since variable vertices are always active, active subgraphs that consist only of rules in R_{const} are defined by G_{const} and are the same for both times: $\forall G_i$ s.t. $R_i \subseteq R_{const}$: $G_i^t = \langle A_i, R_i, E \rangle = G_i^{t+1}$. Since we know that x_i^t is a solution for G_i^t, it must also be a solution for G_i^{t+1}. Let us denote the set of valuations for G_{const} as x_{const}.

Let x_{nc}^{t+1} represent (newly found) solutions for all active subgraphs $G^{t+1} \backslash G_{const}$. As proven in Lemma 1, the combined $x^{t+1} = x_{const} \times x_{nc}^{t+1}$ must be a solution for the $CSP(A, F_A^{R^{t+1}})$. Therefore it is proven that it is possible to reuse solutions x_{const} from G_{const} in a global solution. □

In the case of a usual CSP instead of an optimization CSP, i.e. if the cost of the solution is not relevant and any solution that satisfies the rules is equally good, the dynamic rechecking can be made even smaller, as the rechecking will be required only if the constraint is added (i.e. the rule becomes active), but not if the rule becomes inactive, as in this case the previous solution is still valid.

Algorithm 9 presents the reaction of the system to a new sensor reading $s = d_s$. For all rules from a ruleset R that depend on s, the system checks the status of the rule (*active* vs *inactive*), and if the status is changed, the rule is marked accordingly.

While a *changed* rule r exists, the system finds a set of adjacent active subgraphs for this rule. If rule changed to inactive there may be more than one.

Algorithm 1 Event processing

```
1: function processChange (s,d_s)
2: for all rule ← R s.t. rule.F_s contains s do
3:    Update rule status
4:    Mark rule as changed if status is changed
5: end for
6: while ∃rule ∈ R s.t. rule is changed do
7:    subGraphs ← findActiveSubGraphs(rule)
8:    for all sg ← subGraphs do
9:       newstate ← OptimizeCSP(sg)
10:      for all v_c ∈ sg.vars s.t. v_c ≠ newstate.v_c do
11:         createAction(v_c,newstate.v_c)
12:      end for
13:      Unmark changed status from all r ∈ sg.rules
14:   end for
15:   Unmark changed status from rule, if still marked
16: end while
```

The optimization CSP is invoked for every such subgraph. For all actuators that changed their state an action is created and is sent further to be executed. Finally, the system removes *changed* status from all rules in checked subgraphs and the original rule.

4.5 Evaluation

4.5.1 Architecture

The internal architecture of the Rule Maintenance Engine implementation is shown in Figure 4.2.

The Web User Interface presents all information about the RME system and its decisions to users. The RME itself runs as a back-end server, and provides a REST interface to show and modify the data [Fielding and Taylor, 2002]. The REST interface is used by the front-end of the system, which consists of a client web interface which runs on the Play Framework[1] and an HTML5-based interface for building context information and immediate manual control via mobile devices, such as Android based smartphones and tablets. The REST interface can also be

[1] `http://www.playframework.com/`

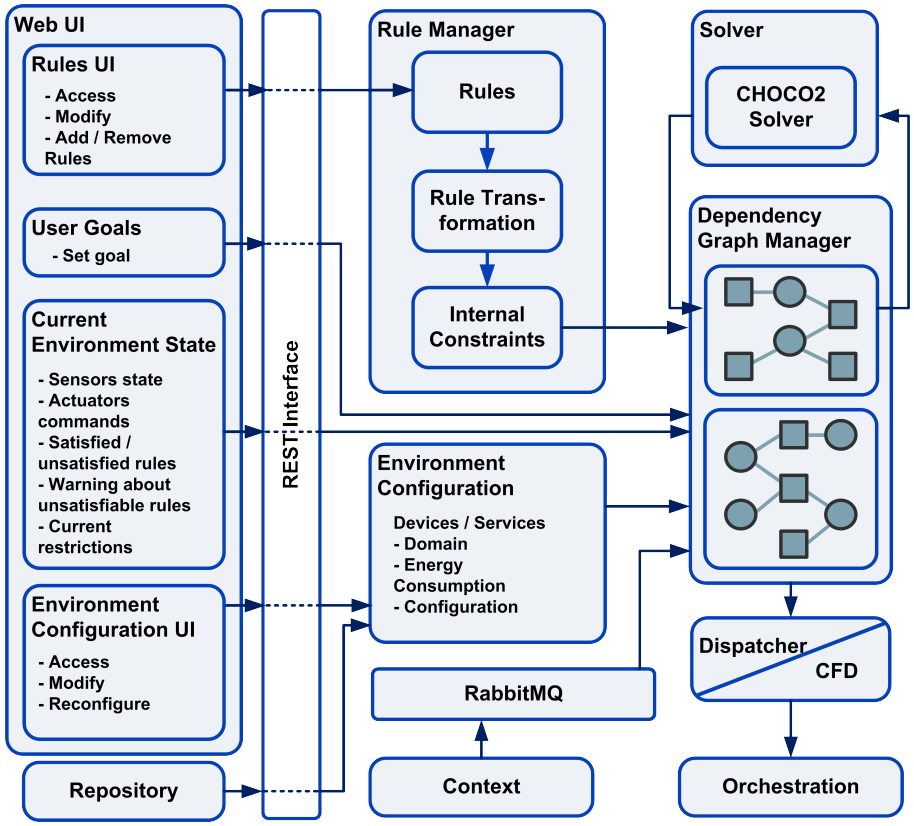


Figure 4.2: Rule Maintenance Engine Architecture

used by other applications to make modifications to the system programmatically.

The initial configuration of the RME system is loaded at startup from the Repository. The Repository contains all required information about the devices, services, or virtual variables, which together form an environment description for the RME. The Repository also contains the latest values of the sensors, so it is immediately possible for the RME to make the initial check of the environment and issue any state goals. This also makes the system tolerant to failures and crashes, as it automatically returns to its latest state after restart. Users can override any part of the environment configuration via the dedicated Web UI. The reconfiguration, as well as addition of new devices or modification of existing ones can be done dynamically, without the need of restarting the system.

Another part of the RME system is the Rule Manager, which manages the set

of rules of the building behavior. The initial set of rules is loaded at the start of the system. It is also possible to switch between sets, and a user can modify any rule, add new ones, or remove obsolete ones through the dedicated Web UI. Every rule is a formula in predicate logic, it can be added in any form by the users, then it will be checked for correctness, consistency and it will be transformed to the internal constraint form, which is used by the Rule Maintenance Engine. The transformation is done once every time the rule is added or changed, and it may result in several internal constraints from a single initial rule.

Devices and rules are combined in the Dependency Graph Manager (DG). It contains the current environment state; the commands, issued to the actuators, and their execution status; warnings about currently unsatisfiable rules; which manual goals were set by system's users previously, etc. All this information is shown on a "Current Environment State" dashboard of the Web UI. This is one of the main dashboards available to users, through which they can control the system and keep track of its status.

Sensors are the main source of events. The Context component collects raw sensor data, processes it, performs activity recognition [Amft and Lombriser, 2011; Wahl et al., 2012], and sends results to the RME. For the RME effectively both low-level sensors and high-level activities are represented through environment variables. Since there can be many events per second, the scalable and highly reliable messaging system is used to transfer this information. For the GreenerBuildings project the RabbitMQ[2] [Videla and Williams, 2012] messaging framework is used. The RME subscribes to the updates it is interested in, and receives them as soon as they are published by the Context. When the event arrives, the Dependency Graph Manager checks, which parts of the environment may be affected by this change, and whether any actuators' states should be rechecked. If this is the case, the Solver is invoked, which finds the new optimal states of affected actuators. The search problem for the Solver is represented as an optimization constraint satisfaction problem (CSP). We use the CHOCO2 Solver[3] library for the task [Jussien et al., 2008].

The second way of obtaining events is from the User Control (UI), where users may set their goals manually. For example, the current rules may say that a temperature in a certain room may be as low as 19 degrees Celsius, but if a user specifies that she wants the temperature to be 22 degrees, the event will be generated and sent to the Dependency Graph Manager.

Finally, when it is calculated that some actuator should perform a certain action

[2] `http://www.rabbitmq.com/`

[3] `http://www.emn.fr/z-info/choco-solver/`

Table 4.1: Living Lab actuators details

Type	Name	Datatype	States/Range	Number of variables
Misc	Blinds angle	Float	-90.0..90.0	3
Misc	Blinds height	Float	0..265	3
TFD	PMV comfort	Float	-2.0..2.0	1
Misc	Table lamp	Boolean	true/false	4
Misc	Light dimmer	Integer	0..1000	10
TFD	Policy	Set	comfort/economy	1
Misc	Screen	Boolean	true/false	4
TFD	Temperature	Float	16.0..27.0	1
			Total number of actuators:	27

or change its state, the goal is generated by the Dependency Graph Manager and is sent further to the GreenerBuildings system for execution.

4.5.2 Living Lab

The system was evaluated in the living lab constructed on the premises of the Technical University of Eindhoven, the Netherlands. In this section we will describe the implementation details of the living lab.

The living lab features two large spaces: a working room with four working desks, and a meeting room, with a meeting table and a presentation area. The sensors include Plugwise power meters[4], CO_2 and humidity, passive infrared (PIR) motion, temperature, light, ultrasound (USR), acoustic. The actuators include Plugwise switches for devices such as projectors and lamps, dimmers for fine-grained control of ceiling lamps' light levels, motor controllers for blinds heights and angles, HVAC system.

As variables, the Rule Maintenance Engine contains both raw sensor data and activity recognition results. Tables 4.1 and 4.2 show the description of the variables within the RME system. In total there are 135 variables. Among them 82 represent physical sensors and contain raw sensor data, 26 represent virtual sensors with higher level recognized activity, and 27 represent actuators, 3 of which belong to the thermo-fluid dynamics (TFD) system.

Rules changed over time, with the original preset having 39 rules that are transformed as described in Section 4.3 into 62 internal constraints. The rules are designed for different adaptation scenarios, which include the adaptation for

[4] `http://www.plugwise.com/`

Table 4.2: Living Lab sensors detailed

Type	Name	Datatype	States/Range	Number of variables
Raw	Projector	Boolean	true/false	1
Raw	CO^2 level	Float	100..3000	1
Raw	Outdoor CO^2 level	Float	100..3000	1
Raw	Computer status	Boolean	true/false	4
AR	Computer work	Boolean	true/false	4
AR	Desk work	Boolean	true/false	4
Raw	Door	Boolean	true/false	2
AR	Energy balance	Float	-10000..10000	1
AR	Heat losses	Float	-10000..10000	1
AR	PMV Status	Float	-2.0..2.0	1
AR	Policy Status	Set	comfort/economy	1
Raw	Humidity	Float	0..100	4
Raw	Outdoor Humidity	Float	0..100	1
Raw	HVAC heat production	Float	-10000..10000	1
Raw	HVAC status	Boolean	true/false	2
Raw	Lamp status	Boolean	true/false	4
Raw	Light power consumption	Integer	0..400	4
Raw	Lux level	Float	0..30000	5
Raw	Outdoor lux level	Float	0..30000	3
Raw	Lights status	Boolean	true/false	4
Raw	Lights switch	Set	0, 1, 2	4
AR	Meeting brainstorming	Boolean	true/false	1
AR	Presentation	Boolean	true/false	1
Raw	Power consumption	Float	0.0..1320.0	15
Raw	Distance	Integer	0..100	9
Raw	Motion	Boolean	true/false	4
AR	Number of people	Integer	0..50	2
AR	Area presence	Boolean	true/false	9
Raw	Status screen	Boolean	true/false	4
Raw	Outdoor temperature	Float	-10.0..80.0	1
AR	Indoor temperature mean	Float	16.0..27.0	1
Raw	Indoor temperature	Float	-10.0..80.0	4
Raw	Window	Boolean	true/false	4
			Total number of sensors:	108

natural and artificial lighting, different activity types in a meeting room, rules for working space personalization, heating system, etc. Control UI allows users to override system's decisions, and set any actuator manually. Here we present the main ideas of several rule adaptation scenarios.

The first scenario concerns adaptation of natural lighting, and contains rules for blinds control (except those that are defined for the meeting and presentation activities, see below). It is usually beneficial for a room to get natural light and warmth from the outside, but when the natural light outside is too bright, it causes glares inside, so the system must ensure that the blinds angle is enough to give sufficient light inside, but not that small to enable sun glare, which decreases people's comfort. The total number of human-defined rules for this case was 5, and these rules translated into 15 internal rules. The examples of these rules are:

$$room313.presence1 = false \Rightarrow room313.blinds.height1 = 0$$

$$room313.presence1 = true \Rightarrow room313.blinds.height1 = 100$$

$$light.luxlevelout1 > 5000 \Rightarrow room313.blinds.angle1 > 70$$

The next scenario is the activities in the meeting room and adaptation to them. It is possible to have a normal brainstorming meeting inside, or to give a presentation, or even none of the above, as the room is open for occasional presence. If people are present inside, but there is no presentation, the total lighting level should be sufficient (at least 500 lux). If the brainstorming meeting is taking place, the light levels should be even higher. There are several different dimmers on the ceiling, so there are many different ways to achieve such conditions. If people use manual control to set certain dimmers to their preferred level, it is possible to use dimmers in other areas to satisfy the rule.

If the presentation is in progress, special lighting conditions should follow. First of all, the dimmable light spot directly above the presentation screen should be fully off, otherwise the visibility of the screen severely decreases. Other dimmers should keep certain level of light, which, however, should stay low, so not to decrease the screen visibility. The 3 human-defined rules are translated into 10 internal rules. Examples are:

$$room313.presence1 = true \wedge room313.meeting.presentation \neq true \Rightarrow room313.light.dimmer1 > 200 \wedge room313.light.dimmer2 > 200 \wedge room313.light.dimmer4 > 100$$

$$room313.meeting.brainstorming = true \wedge room313.meeting.presentation \neq true \Rightarrow room313.light.dimmer2 > 600 \wedge room313.light.dimmer3 > 600 \wedge room313.light.dimmer4 > 600$$

$$room313.meeting.presentation = true \Rightarrow room313.blinds.angle1 > 80 \wedge room313.light.dimmer2 = 0 \wedge room313.light.dimmer1 < 300 \wedge room313.light.dimmer4 < 300$$

The heating mechanism of GreenerBuildings is based on the Computational Fluid Dynamics (CFD) module[5], which calculates the air quality, temperature, humidity, and other climate conditions within the room, and uses available actuators for fine-grained control of the climate comfort levels. As there is a dedicated module which does the required complex computations, the RME does not invoke heating actuators (such as HVAC module) directly, and instead has abstracted actuators, namely mean temperature, policy (economy or comfort) and PMV (predicted mean vote) comfort level. When the best value of abstracted actuators is calculated, it is sent to the CFD component, which performs the necessary fine-grained control of physical devices. The RME rules include having a comfort policy only when there are people inside (otherwise it is automatically set to economy), and stopping all fine-grained control if windows are open. Most of the time, people add their own rules for the temperature levels, which they deem comfortable to them, so the temperature rules are not included in the preset. The 2 rules that are included correspond to 5 internal rules:

$$room313.window1 = true \vee room313.window2 = true \Rightarrow room313.pmv = unmanaged \wedge room313.temperature\_mean = unmanaged$$

$$room313.presence1 = true \Rightarrow room313.policy = comfort$$

Working areas are the areas where most of the "personalized" rules appear, as naturally the area is owned by one person. The preset rules use standardized approach, by turning off the screen, when the person is not around, distingushing the lighting conditions for desk work and computer work, etc. The human-defined preset rules include 17 rules, which translate to 20 internal rules. Examples are:

$$room326.presence1 = true \Rightarrow room326.screen1 = true$$

$$room326.desk1.deskwork = true \Rightarrow room326.light.dimmer1 > 700$$

$$room326.presence1 = true \wedge room326.desk1.computerwork = false \Rightarrow room326.light.dimmer1 > 500$$

$$room326.desk1.computerwork = true \Rightarrow room326.light.dimmer1 < 500$$

There is a special type of rules, introduced for smooth continuous operation of the system: trigger rules. The initial reason for them is the capability of the people

[5]Fluid Solutions - @lternative, `http://fluidsolutions-a.com/`

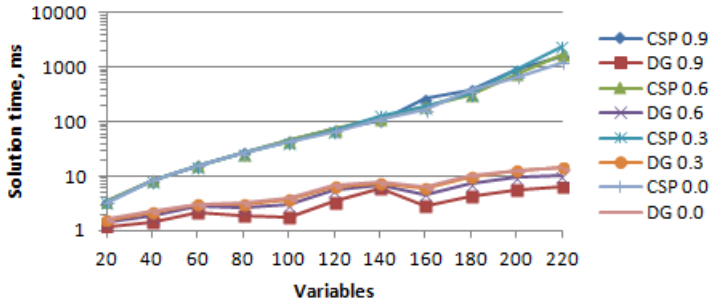
inside the building to manually control some of the actuators. For example, lamps or the temperature setting is generally controlled by the system via the defined rules. However, a person has the ability at any moment to press the lamp switch or to set a certain temperature level on the thermostat. Such ability to manually set the state of the building even in contradiction with previously defined rules is very important to retain even in highly automated buildings, as it keeps people in full control increasing their overall satisfaction level. So, once a person sets some actuator manually, this actuator should be forbidden to be changed by the system. It may still be possible to satisfy existing rules in a different manner. For example, for the rule (4.1), if a person prefers to keep the blinds closed, it is still possible to turn on the lamp, which will satisfy the rule. But then another problem arises. Once a certain actuator is activated manually, when is it possible for the system to "take it back", i.e. to remove the restriction on its state modification? For example, once a person turned on a lamp, if she goes away from the room, we may assume that the restriction is no longer valid. Trigger rules are designed specifically for such case. The rule may be specified, as usual, but it must be specifically marked as a trigger rule, and two additional things must be provided: (1) the actuator, for which the restriction is removed once the rule is satisfied, and (2) the type of the restriction that is removed. The current implementation of the system supports two types of restriction: the manual user input, and the error in the device (which marks the device as broken, and stops the control of it until fixed). In the preset, all devices are released from their user restriction if the room becomes unoccupied. There are 12 preset trigger rules, an example of such a rule is:

$$\begin{aligned} &actuator: \ room326.light.dimmer3 \\ &type: \ user\_feedback \\ &rule: \ room326.presence1 = false \end{aligned}$$

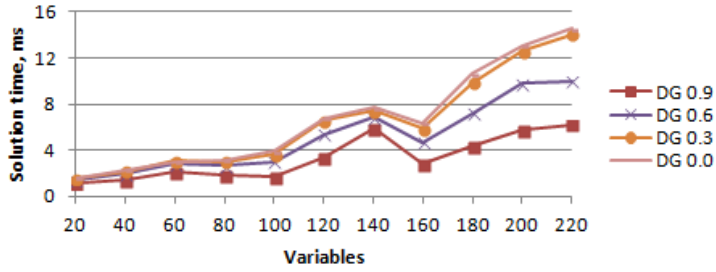
The operation of the living lab showed that our module solves all resulting CSPs in a matter of milliseconds, returning real-time commands to actuators. The next step of the project is to extend the system to more rooms, and the whole building, so the next section discusses the performance and scalability potential of our solution in depth.

4.5.3 Performance

To evaluate the effectiveness of our solution with greater flexibility, we also made performance experiments that were running on Windows 7, Intel Core2Duo E7400 @2.8GHz, 4 Gb RAM, Java7 machine. As a baseline, we used random instances with boolean variables. Note that any instance with arbitrary sizes of domains



(a) CSP vs. DG, log scale, clusterization values of 0.9, 0.6, 0.3, 0.0



(b) DG-only close-up

Figure 4.3: Average solution times of CSP and DG representations

can be converted into an equivalent instance with boolean variables, one per each domain value. Every instance has half of its variables as sensors, and half as actuators. For every set of parameters we generate 50 different instances. Every instance ran for 100 sensor change events. For every event the time to find a solution is recorded, and the average time across these runs is presented in the figures. Every rule is a random constraint between two sensors and two actuators, and the number of rules equals to the 120% of the number of variables.

We also analyzed the impact of clusterization on the performance of the DG solution. In smart environments most variables are naturally split into clusters of highly-dependent variables, e.g. by location, with loose dependency between clusters. Thus we introduce clusters of variables in our instances, with varying degrees of clusterization. For example, for a degree of 0.6, 60% of rules will connect variables within a cluster, and remaining rules connect any variables, also across clusters. We used clusterization values of 0.9 (very distinctly defined clusters), 0.6, 0.3 and 0.0 (no clusters, every rule connects variables fully randomly). The number of clusters is $\sqrt{|V|}$, so an instance with 40 variables has 6 clusters with 6-7 variables

each, while an instance with 400 variables has 20 clusters with 20 variables each.

Figure 4.3 compares solution times using a natural CSP definition (as given in Section 4.2), and using the Dependency Graph data structure. The time of rule activeness rechecking and graph traversals is *included* into the resulting time for the DG, i.e. results include all overhead, associated with using the DG data structure. It can be seen that for all cases DG severely outperforms the natural CSP definition, staying at around 10 milliseconds time for over 200 variables, while CSP already goes to over 1000 milliseconds solution time for such cases. The clusterization parameter has no influence on CSP solution time, which is expected, since CSP takes the full environment into account. However, for DG it is shown, that the bigger the clusterization is, the lower the solution time will be, which also means much bigger scalability potential for implementing the solution in smart buildings.

Table 4.3: Random instance run, 100 variables, 0.0 clusterization

Event		1	2	3	4	5	6	7	8
CSP Time		45.68	41.38	71.06	36.05	25.24	32.93	34.02	66.47
DG	**Time**	8.71	12.11	8.47	10.62	7.86	6.71	5.69	4.19
	Size(s)	1;21	22;1;1;8	1;1;20	26;2	3;25	25;5	1;26	1;1;3

Event		9	10	11	12	13	14	15
CSP Time		29.42	31.18	31.04	56.98	29.20	29.10	66.16
DG	**Time**	8.33	7.83	4.88	0.08	18.04	2.80	4.46
	Size(s)	1;1;23;1	22;1;2;1	2;20	-	2;1;2;1;1;20	21	3;5;17

For better insight we included the detailed data from one of the runs of the system on an instance of 100 variables with 0.0 clusterization in Table 4.3. Every event corresponds to a single sensor change. The size of the CSP definition is always the same (100 variables, among which 50 are decision variables, i.e. actuators), while the DG size varies, depending on the current size of active subgraphs. As every sensor can be a part of several rules, it is customary that a single sensor change triggers re-optimization of several subgraphs. E.g. event 6 triggers two DG tasks, one with 25 variables, and the other with 5 variables. Event 12 has no impact on active subgraphs, so no re-optimization occurs.

Chapter 5

Interpretation of Inconsistencies via Context Consistency Diagrams

The ability of pervasive context-aware systems to perform efficiently fully relies on the ability to obtain the most detailed, specific, and correct information about the environment.

However, before applications may use the information to make appropriate decisions and adjust their behavior, several steps are required to obtain context information in a proper form. First of all, raw sensor readings should be gathered by a system's middleware from surrounding sensors. Then they should be pre-processed, converted to a logical form, and combined together to obtain a view of the current environment. Afterwards the information should be converted to a form that can be understood by applications.

Several challenges arise during this process. The sensors are often noisy, imprecise, and their readings are easily corrupted, which may lead to inconsistencies and conflicts in gathered data. Also, the full information about the environment is practically impossible to obtain. Some portions of the environment can not be physically read by given technology, and there is always something happening that sensors miss to detect, e.g. [Jeffery et al., 2006] showed that in dynamic environments the percentage of correctly read RFID tags may be as low as 60-70%. Another issue of sensor readings gathering is that information becomes obsolete rapidly. The data that was correct at the time of reading may be already obsolete when it reaches the system's middleware and gets processed. The asynchronous nature of sensor readings leads to alterations in the order of readings' arrival to the middleware. Finally, the automated processing of sensor readings into an interpretation of the environment may introduce errors itself. Classical examples of such errors are image recognition mistakes.

In the presence of a conflict among sensor readings, the conventional research [Bu, Gu, Tao, Li, Chen and Lu, 2006; Xu et al., 2008] suggests to discard one of the readings that is deemed as incorrect based on some heuristic strategy. Different heuristics have been proposed, among which the removal based on relative fre-

quency [Bu, Gu, Tao, Li, Chen and Lu, 2006], drop-latest, drop-oldest, drop-all, or drop-random [Xu et al., 2008] strategies.

Such a removal is usually done as soon as a conflicting sensor reading is received, to keep the full interpretation of an environment without conflicts. The removal of sensor readings in an ambiguous situation may, however, cause even more problems, in case a correct sensor reading is removed instead of an incorrect one. A more cautious approach that removes all conflicting sensor readings may drastically reduce the available amount of information, which is used by high-level applications to make decisions.

In this chapter, we propose a mechanism of reasoning about sensor information to define possible context interpretations. This includes both the ability to reason about a context with incomplete knowledge, as well as the ability to cope with erroneous contexts that may lead to false beliefs. We propose a data structure called *context consistency diagrams (CCD)* that allow to efficiently represent acquired context information together with all possible context inconsistencies and interpretations. CCDs can be efficiently maintained and queried in real-time, and can be used to obtain information about the likelihood of particular context interpretations, sensor values or relations between sensors.

5.1 System model

Context-aware reasoning systems are complex software that produce an application-friendly interpretation of given raw sensor data. A possible high-level architecture for such systems is given in Figure 5.1. Often, an optional rule-based pre-processing of raw sensor data is performed. In Figure 5.1, a rule ($TVsound = max \implies TVchannel \in \{sports, shows\}$) is applied to a sensed value ($TVsound = max$). The resulting pre-processed context ($TVsound = max, TVchannel \in \{sports, shows\}$) is then passed to a context subsystem ("CCD representation" layer, see Section 5.2) that is responsible for efficient storage of acquired context information, resolving inconsistencies, answering to queries, or triggering events to the subscribed top-level applications.

To deal with inconsistencies, we use *context consistency diagrams*. A CCD is considered *inconsistent* if there is no single interpretation that is confirmed by all sensed (and then pre-processed) data. Such conflicts are caused by sensor imprecision, incomplete, missed or obsolete data. Ideally, conflicts caused by a failed sensor or by data expiration should not stop the system from providing the best possible interpretation for the acquired context information. Several techniques exist to resolve an ambiguous conflict in favor of one interpretation. But if the res-

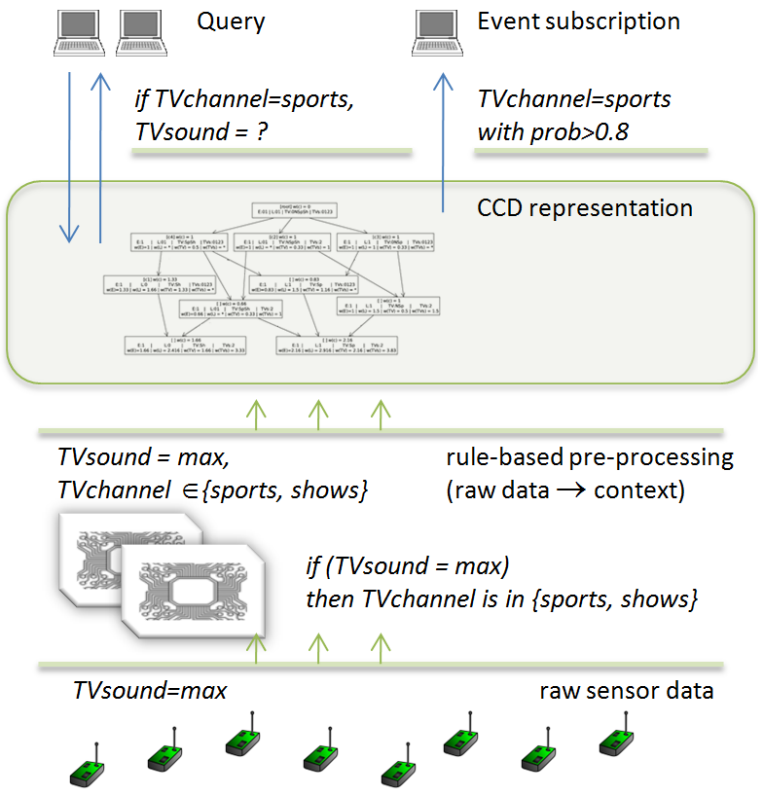


Figure 5.1: Context reasoning using CCD

olution is incorrect, further interpretations of a situation will also be wrong, even if further information may show that another solution was preferable. To deal with this, a CCD keeps several interpretations, each with its own probability of being true.

To resolve a conflicting context, we associate a likelihood of a certain context information to be true to each acquired chunk of data. Whenever several portions of contexts "support" each other (that is, there is an interpretation of a situation that is consistent with all of them), their mutual truth likelihood is higher comparing to the conflicting ones. Additionally, each arrived pre-processed context information is sharing a certain degree of truth likelihood, thus compensating the effect of a faulty sensor over the inferred information received from that particular sensor. The most probable interpretation is then the one that is "supported" by the majority

of individual contexts.

Even if a particular context does not support the most "popular" interpretation, it is still stored in a CCD. Thus a CCD query may return several interpretations, each with its own probability of being true. It might happen that with the acquisition of new sensor data, another interpretation is considered more likely, if the new data support it.

Note that with such structure the context interpretation is never final, as new data may change the interpretation by contributing to an interpretation previously considered wrong. However, explicit description of different interpretations can grow in space exponentially to the number of found inconsistencies. In this case, CCD reasoning should discard contexts that support the most "unlikely" interpretations, as, most probably, they represent faulty or imprecise sensors.

5.2 Context consistency diagram

5.2.1 Context

A server that processes contexts obtains information from the underlying layer, shown in Figure 5.1 in a form $v_i = d$, i.e., a variable v_i has value d. More precisely,

Definition 5 (Environment)**.** An *environment* is defined by a set of context variables $V = \{v_1, v_2, ..., v_n\}$. Each variable v_i varies over a domain $D_i = \{d_{i1}, d_{i2}, ..., d_{im_i}\}$ with size m_i.

It is also possible that the sensors (pre-processing layer) return a range of values, i.e., $v_i \in \{d_j\}$. For example, a location variable may be sensed by many location sensors (e.g. RFID are known to be imprecise).

However, many variables either cannot be directly observed, or can only be partially sensed. If the heating mechanism is broken, we sense that the heater was turned on, but we cannot observe if it has actually started to heat the room, unless we have a temperature sensor. Fortunately, many variables influence each other. For example, it is impossible to have a light turned on, if there is no electricity in the house; a location of the person and a location of the tool that she works with must be in the same room, etc. If these correlations are taken into account, even a few observed variables may give an overall (yet possibly incomplete) knowledge about the environment.

Definition 6 (Context, Interpretation)**.** For a given environment $\langle V, D \rangle$, a *context* c is a valuation of all variables in V with a non-empty subset D^c of D. If all

Table 5.1: Variables example

Variable	**Domain**
Electricity	off, on
Light	off, on
TV	off, news, sports, shows
TV sound	0, 1, 2, 3

Table 5.2: Dependency rules example

$\neg(E = off \wedge (L = on \vee \neg(TV = off)))$	Light and TV can be turned on only if electricity is on.
$\neg(TV = off \wedge TVs \in \{1, 2, 3\})$	Non-silent TV sound means TV is turned on.
$TV = shows \Rightarrow L = off$	If TV channel is shows, light should be turned off.

variables v_i are assigned one and only one specific value in D_i, a context is called an *interpretation*.

Non-emptiness of a subset D^c ensures that a context is always possible in practice, i.e. each variable has at least one possible value.

We represent a context by enumerating possible context variables' values: $\{D_1^c, D_2^c, \ldots, D_n^c\}$, or, alternatively, as $v_i \in \{d_{il}, \ldots, d_{ik}\}$. We write $c.v_i$ to refer to i-th variable of context c.

Our knowledge about the environment is described by a set of contexts $\{c_1, ..., c_n\}$. If for any two interpretations x, y such that $\forall c_i : x \in c_i \wedge y \in c_i$, it follows that $x = y$, then we have *complete* and *unambiguous* knowledge about the given environment. More than one interpretation represents an ambiguity or incomplete knowledge of the environment. Intuitively, each new sensor reading adds some more knowledge about the environment, thus it reduces the number of possible interpretations. Faulty contexts can be detected when an impossible situation is created, i.e. when there is no interpretation x, such that $\forall c_i : x \in c_i$.

In Table 5.1 a portion of a smart home is modeled by four context variables. In Table 5.2, few pre-processing rules are defined that represent the inter-relation between the context variables. Note though that it is not important how these rules are defined, as far as they result in a context information (similar to the

one shown in Table 5.3). The first two rules represent basic physical laws: there must be electricity in the house for the light and TV to be turned on, and the TV volume must be higher than null if the TV is turned on; and the third rule is set specifically by a smart house's resident: if the channel is set to 'shows,' the room's light should be turned off. Using these rules, from a single reading that the light is on, we infer that the electricity is on, and if the TV is on as well, the channel is definitely not 'shows.'

A set of contexts $C = \{c_k\}$ is *consistent* if there exists at least one interpretation $x : x.v_i = d_{ij_i}, \forall i \in 1..n$ such that $d_{ij_i} \in c_k.v_i,\ \forall c_k \in C,\ \forall i \in 1..n$. A set of contexts is *inconsistent* otherwise.

Additionally, we define two relations over contexts:

- *Inclusion*: $c_1 \subset c_2$ iff $\forall i \in 1..n :\ c_1.v_i \subset c_2.v_i$ Inclusion can be viewed as a relation of a more precise and less precise contexts. If $c_1 \subset c_2$ then context c_1 is more precise than c_2, in other words, each variable of c_1 contains less values that are possible.

- *Intersection*: $c_u = \bigcap_{j=1}^{k} c_j = c_1 \cap c_2 ... \cap c_k$ iff $\forall i \in 1..n :\ c_u.v_i = c_1.v_i \cap c_2.v_i ... \cap c_k.v_i$ An intersection of inconsistent contexts always equals to $\varnothing$. An intersection of consistent contexts is a context, that is at least as precise as any of the originals: $\forall j \in 1..k\ \ c_u \subseteq c_j$.

5.2.2 Context consistency diagram

To compactly represent all possible interpretations for a given set of contexts, we use relations defined in the previous section, thus forming a diagram with arrows representing the inclusion relation. Any two contexts c_i, c_j are connected in the diagram if $c_i \subset c_j$, and there is no such c_k such that $c_i \subset c_k \subset c_j$.

The idea of putting contexts into the diagram structure is essentially an introduction of a compact representation of all possible interpretations of the environment. The "full domain" context is always at the top, meaning "no information is known; any situation is possible". Starting from the top and going down, contexts become more and more precise, with the most restrictive (as well as the most knowledgeable) contexts at the bottom. Formally, CCD is defined as follows:

Definition 7 (Context consistency diagram (CCD))**.** Given an environment $\langle V, D\rangle$ and a set of contexts $C_0 = \{c_k\},\ k \in 1..N$, a *context consistency diagram (CCD)* is a tuple $\langle C, E, r\rangle$, where:

- $r = D$, is a special context, the *root;*

Table 5.3: Example of sensor readings and contexts

ID	Sensor reading	Context
c_1	$TV = Sh$	$E:1 \mid L:0 \mid TV:Sh \mid TVs:0123$
c_2	$TVs = 2$	$E:1 \mid L:01 \mid TV:NSpSh \mid TVs:2$
c_3	$L = 1$	$E:1 \mid L:1 \mid TV:0NSp \mid TVs:0123$
c_4	$TV \in \{Sp, Sh\}$	$E:1 \mid L:01 \mid TV:SpSh \mid TVs:0123$

- $C = C_0 \cup C_u \cup r$ where C_u is the full set of intersections of a power set of C_0.
- $E \subseteq C \times C$, such that $(c_2, c_1) \in E$ iff $\exists c_1, c_2 \in C : c_1 \subset c_2$ and $\nexists c_m \in C : c_1 \subset c_m \subset c_2$.

Contexts from a set C are vertices of the diagram and E is a set of directed edges. In a relationship $(c_1, c_2) \in E$, c_1 is called a **parent**, and c_2 is called a **child**. c_p is called a **predecessor** of c_c, and, respectively, c_c is called a **descendant** of c_p if either of the following holds:

1. $(c_p, c_c) \in E$
2. $\exists \{c_i\} \in C, i \in 1..k \; s.t. \; (c_p, c_1) \in E \wedge (c_k, c_c) \in E \wedge (c_i, c_{i+1}) \in E, \forall i \in 1..k-1$

We write $\psi(c)$ to denote the full set of descendants of c and $\Psi(c)$ to denote the full set of predecessors of c. Several important characteristics of the CCD directly follow from its definition:

1. An intersection of two consistent contexts $c_1 \in C$ and $c_2 \in C$ is a descendant of both contexts. $\exists c_u \in C, \; c_u = c_1 \cap c_2 \; s.t. \; c_u = \psi(c_1), \; c_u = \psi(c_2)$.

   If $c_1 \in C$ and $c_2 \in C$ are inconsistent, then they do not have common descendants. $\nexists c \in C \; s.t. \; c = \psi(c_1), \; c = \psi(c_2)$.

2. If a set of contexts is empty, then CCD has only one root context. $C_0 = \varnothing \Rightarrow G = \langle r; \varnothing; r \rangle$

3. There is no context that is a predecessor of the root. A root is a predecessor of all other CCD contexts. $\forall c \in C : \nexists (c, r) \in E \; , \; r \in \Psi(c)$

For a set $C_0 = \{c_1, c_2, c_3\}$, the corresponding set of intersections of its power set is equal to $C_u = \{c_1 \cap c_2, c_1 \cap c_3, c_2 \cap c_3, c_1 \cap c_2 \cap c_3\}$.

For a set of contexts listed in Table 5.3 the corresponding CCD is shown on Figure 5.2.

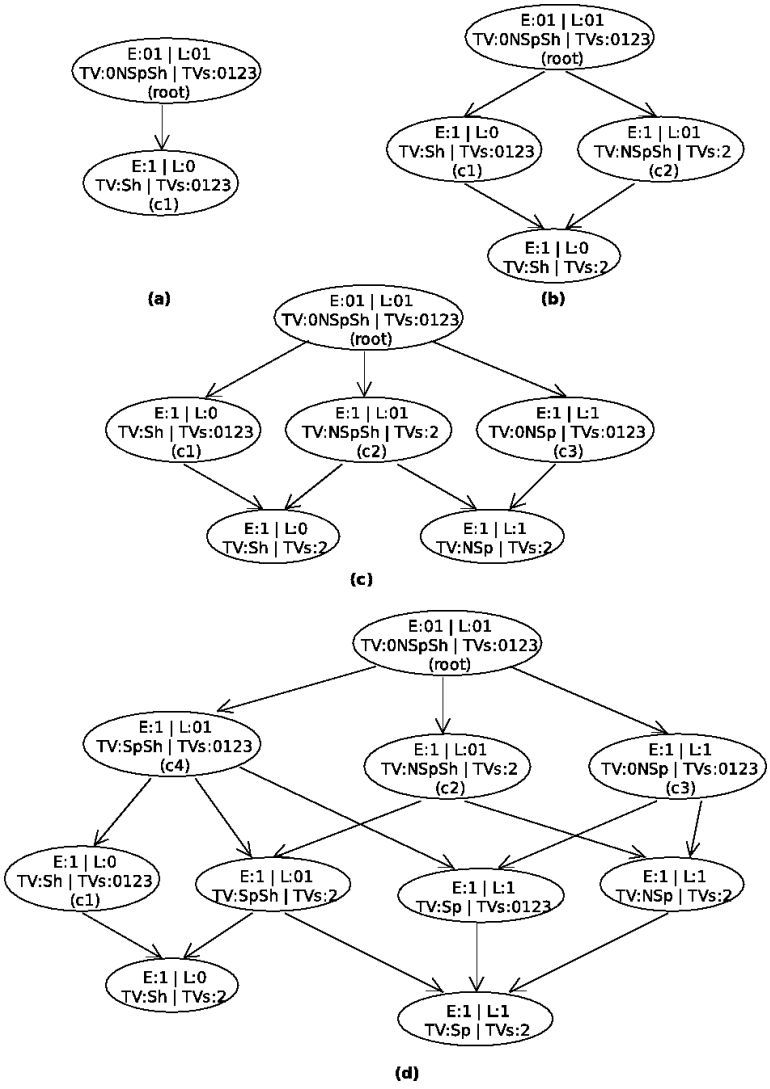


Figure 5.2: Example of context consistency diagrams

We assume that all variables in a context are dependent on each other (Section 5.2.1). For independent variables, we can split a context into non-intersecting subgroups and, therefore, we instead produce the CCDs of smaller sizes for each subgroup.

5.3 Calculation of probabilities

When the CCD results in more than one interpretation, it is important to assess the likelihood of each interpretation. For a query (Figure 5.1), we provide answers for the following three possible requests:

1. The probability that a particular situation is true.
2. The probability that a variable has a certain value.
3. The dependency of variables on one another. In other words, the conditional probability that a certain variable has a certain value in case another variable has an *a priori* known value.

We now describe how the CCD is used to address all above queries at any given moment. To calculate the probabilities mentioned above, we first need to introduce the concept of initial weight function $w_0(c)$. The initial weight function shows the importance of each original context $c \in C_0$. The weights depend on many things, among which are the infrastructure of sensor network; the importance of each sensor (the more important is the sensor, the more important is the context, associated with the sensor reading); and the number of times a particular context has been read. The set C_0 contains only unique contexts, but if one context was read two times (by two different sensors, or by the same sensor at subsequent time steps), usually it should be regarded as more important than the one that was read only once. By default, or when the initial probabilities are unknown, we assume a uniform distribution, that is, any sensed information is equally likely. In the presence of additional information, other strategies for assigning weights may be chosen. The strategy for assigning weights should be chosen at the initial setup. The example in Figure 5.3 assigns the weight 1 uniformly to all contexts.

For all contexts that are not in C_0 the initial weight function equals to 0:

$$\forall c \notin C_0 \ : \ w_0(c) = 0$$

The full weight function $w(c)$ for each context in a CCD is defined as

$$w(c) = w_0(c) + \sum_{\forall c_p \in \Psi(c)} w_0(c_p)$$

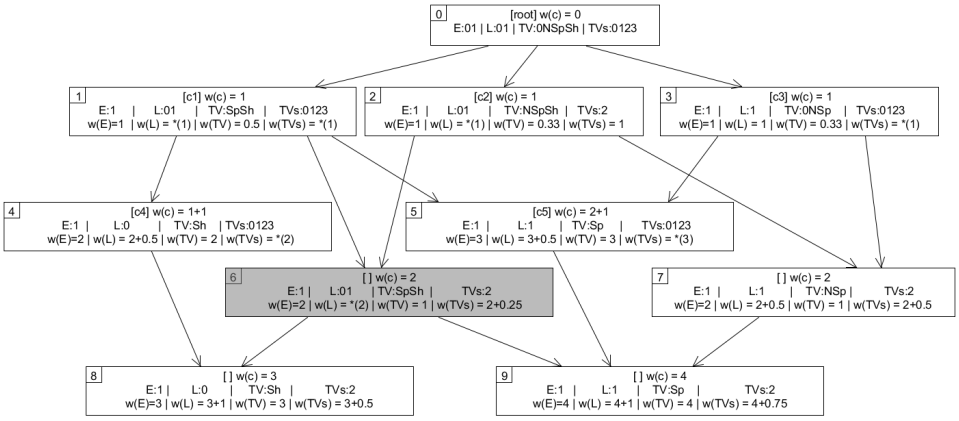


Figure 5.3: Assigning weights to the CCD

The full weight function takes into account that contexts that are consistent with each other should weigh more than inconsistent ones. The idea is that consistent contexts form a consistent view on the situation, thus they all can be correct. But in the set of inconsistent contexts some are certainly faulty. So the full weight function rewards contexts for being consistent with others by increasing the weight of their descendants.

The full weight of the CCD is the sum of weights of all its contexts:

$$w(G) = \sum_{c \in C} w(c)$$

To calculate the probability that a variable has a certain value, we adapt the weight of the context to calculate the weight of each value of the variable inside a context. The context with several values of some variable assumes that each of these values is equally probable, so we divide the weight of the context among all values for each variable:

$$w(c.v_i = d_{ij}) = \frac{w(c)}{|c.v_i|},\ \forall d_{ij} \in c.v_i$$

However, we do it only if the context actually knows something about the variable v_i. For example, if we got a sensor reading that the light is on, it tells nothing about the TV sound, so we do not split the context weight among TV sound values. But if later we receive a context that tells us both that the light is on and TV sound is 2, then the first context *supports* the second one (since they are consistent), so we transfer the weight of the TV sound of the first context to the second one. For this we introduce the $\star$ value for a variable weight. This value means that the weight

is transferred to the children of the context. Taking this into account, the weight of each value of each variable in a context is given by the following formula:

$$w(c.v_i = d_{ij}) = \begin{cases} \star & \text{if } c.v_i = D_i \\ \frac{w(c)+t(c.v_i)}{|c.v_i|} & \text{otherwise} \end{cases}$$

where $t(c.v_i)$ is a *transfer* (or *carrying*) value from the parents of the context:

$$t(c.v_i) = \sum_{\forall c_p \in \Psi(c) \ \& \ w(c_p.v_i)=\star} \frac{w_0(c_p) \times |c.v_i|}{|c_p.v_i|}$$

$\star$ also contributes towards the efficiency during CCD updates. To calculate the final probabilities, we treat $\star$ differently depending on the context having children or not. If the context has children, the weight of a variable is fully transferred to them. Otherwise, we have no knowledge whatsoever about the value of the corresponding variable, so each domain value gets equal share of the full weight:

$$(w(c.v_i) = \star) \Leftrightarrow \begin{cases} w(c.v_i = d_{ij}) = 0 \\ \qquad \text{if } \exists (c, c_c) \in E \\ w(c.v_i = d_{ij}) = \frac{w(c)+t(c.v_i)}{|c.v_i|} \\ \qquad \text{if } \nexists (c, c_c) \in E \end{cases} \qquad \forall d_{ij} \in D_i$$

Now we can calculate the probability for each variable that it has a certain value:

$$pr(v_i = d_{ij}) = \frac{\sum_{\forall c \in C} w(c.v_i = d_{ij})}{w(G)}$$

As an example, we describe the calculation of weights on context 6 (grayed out) in Figure 5.3. The full weight of the context is 2, and is obtained from its two parents $w(1)$ and $w(2)$. This weight fully goes to the sole value of the variable E, so $w(E = 1) = 2$. But for values of the variable TV, this weight is split equally in two, so each value $TV = Sp$ and $TV = Sh$ gets half of the full weight, or 1. The weight of the variable TVs is equal to 2.25, because it combines the weight of the context 6, and a fourth part of a transfer value from the context 1. The weight of the variable L is equal to $\star$,because this variable allows any value of the corresponding domain. So, this context does not assign any weight to values of L, but instead transfers it to its two children.

If we want to calculate the conditional probability that a certain variable has a specific value in case another variable has a particular value $pr(v_i = d_{ij}/v_c = d_c)$, we need to reduce weights in a CCD in such a way that only contexts that are

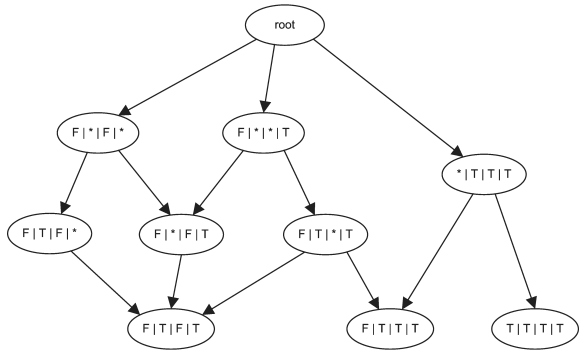


Figure 5.4: CCD example. Every node is in [LCD|PC|PIR|PR] format

compatible with $v_c = d_c$ have weights higher than 0. Also, for contexts that allow other values for v_c we need to correspondingly reduce their weight.

The conditional weight of each context is equal to

$$w'(c) = \begin{cases} \frac{w(c)}{|c.v_c|} & \text{if } d_c \in c.v_c \\ 0 & \text{otherwise} \end{cases}$$

Similarly, the conditional weight of each variable is

$$w'(c.v_i) = \begin{cases} \frac{w(c.v_i)}{|c.v_c|} & \text{if } d_c \in c.v_c \\ 0 & \text{otherwise} \end{cases}$$

Finally, conditional probability is equal to

$$pr(v_i = d_{ij}/v_c = d_c) = \frac{\sum_{\forall c \in C} w'(c.v_i = d_{ij})}{w'(G)}$$

where $w'(G) = \sum_{c \in C} w'(c)$

5.3.1 CCD Example

A portion of environment is modelled by four boolean variables with different weights: PC with weight 1, LCD with weight 1, PIR for (K)eyboard with weight 0.7, and (PR)essure on the chair with weight 0.8.

We need to establish rules on their dependency. First of all, the monitor cannot be turned on if the PC is off. Next, if the PIR near the keyboard detects movement, it must be typing or moving a mouse, and this means that the chair is occupied. Finally, the monitor is designed to turn off after a minute of inactivity, so it's only

active if someone is present and actively works with PC (thus uses keyboard or mouse). So the rules are the following:

$$LCD = true \Rightarrow PC = true$$

$$PIR = true \Rightarrow PR = true$$

$$LCD = true \Rightarrow PIR = true \wedge PR = true$$

When we receive a sensor reading, we apply rules to it to obtain an extended context. For example, if we receive a reading $PIR = false$, the context after all rules are applied is $\{LCD = false; PIR = false\}$. Sometimes it is possible to have more than one context, for example for the reading $LCD = false$ we still need to account for the second rule, which gives us two contexts: $\{LCD = false; PIR = false\}$ and $\{LCD = false; PR = true\}$.

Figure 5.4 shows the CCD constructed if the following four sensor readings are received: $LCD = true; PC = true; PIR = false; PR = true$. We have three possible situations, but they all have different weight. The $[F|T|T|T]$ has weight 1.8, while $[F|T|F|T]$ has weight 2.5, and $[T|T|T|T]$ has weight 2.8, which means that most probably the reading $PIR = false$ was an incorrect one.

5.4 Maintaining CCD

Next, we describe the algorithms for maintaining the CCD while new sensor data arrives. We start by introducing a few properties of the CCD that make the maintenance possible.

Property 1. For a given set of contexts there is one and only one non-isomorphic representation of its CCD.

It follows directly from the rules of construction. C_u is only dependent from C_0, and the root is always the same for the same variables and theirs domains. So $C = C_0 \cup C_u \cup r$ is always the same for the same C_0. For each pair of contexts in the CCD $c_1, c_2 \in C$ we use the rules 2 and 4 to determine if they are connected (one is a parent and the second is a child) or not, i.e. if $\exists(c_1, c_2) \in E$. So, for each C_0 there is only one way to construct a tuple $G = \langle C; E; r \rangle$.

The actual context information changes rapidly and the CCD should be updated in real-time to always conform to it. Sensor readings arrive independently, and the CCD must be reconstructed to accommodate new information. After some time, obsolete contexts should be removed from the CCD to eliminate obsolete situations.

Obviously, the CCD should not be constructed from scratch with each change in a contexts set. Instead, with an arrival of a new context it should be added to the existing diagram (by only changing the affected nodes), and when the context becomes obsolete, it and other obsolete nodes must be removed without affecting other parts of the diagram.

From the fact that for the given set of contexts there is only one CCD follow two more important properties:

Property 2. The order of contexts addition does not change the resulting CCD.

According to this property we can handle contexts updates one by one, without taking into account the order of their arrival, which can vary for asynchronous updates.

Property 3. Adding and then removing a context does not change the resulting CCD.

The properties 2 and 3 follow from the fact that the set C_0 is not ordered.

Algorithm 2 contains the pseudocode of the addition of a new context to the diagram. It is started by running $AddContext(context, root, weight)$ (trying to add a new context directly under the root) and recursively descends to check all contexts that are consistent with a new one.

The function $AddContext(context, parent, weight)$ is called only when a *context* should be a descendant of a *parent*. Firstly it checks if a *context* is already present as a child of a *parent* (lines 3-5), if it is a child of a child (lines 6-8), or if some existing children of a parent should become children of a new context (lines 9-11).

If the *context* is not yet present and not a child of a child, then it is added as a new child, and all existing children are checked for consistency with it. If they are consistent, their intersection is created and recursively added to both contexts (lines 15-21). Note that an intersection context receives no initial weight.

Algorithm 3 shows the removal of an outdated context from the diagram. CCD has to be changed only if there are no other similar contexts and if it has only one parent (line 3), otherwise it must stay in the diagram as an intersection of its parents.

The reduction of the CCD starts with removing a link from a *parent* to a *context* (line 5) and from a *context* to all its children (line 7). We want all children of the removed *context* to be added to its *parent* directly. But we do not want to add a link from a *parent* to a *child*, if they are already linked through different path. So on lines 8-10 we check if this is the case, and if not, we add a link (line 9).

To be sure that no child without initial weight is left with a single parent, we recursively check all children of a context for deletion (line 11).

Algorithm 2 Adding context to CCD

```
1: function AddContext(context, parent, weight)
2: for all child ∈ parent.children do
3:    if child = context then
4:       W_0(child) ← W_0(child) + weight
5:       return
6:    else if context ⊂ child then
7:       AddContext(context, child, weight)
8:       return
9:    else if child ⊂ context then
10:      Remove link from parent to child
11:      Insert link from context to child
12:   end if
13: end for
14: Add link from parent to context
15: for all child ∈ parent.children\context do
16:   if isConsistent(context, child) then
17:      x ← context ∩ child
18:      AddContext(x, child, 0)
19:      AddContext(x, context, 0)
20:   end if
21: end for
```

Algorithm 3 Removing context from CCD

```
1: function RemoveContext(context, weight)
2: W_0(context) = W_0(context) − weight
3: if W_0(context) = 0 and |context.parents| = 1 then
4:    parent ← context.parents
5:    Remove link from parent to context
6:    for all child ∈ context.children do
7:       Remove link from context to child
8:       if ∄brother ∈ child.parents s.t. brother ⊂ parent then
9:          Add link from parent to child
10:      end if
11:      RemoveContext(child, 0)
12:   end for
13: end if
```

Algorithm 4 Retrieve probabilities

1: **function** RetrieveProbabilities(*conditions*)
2: Set all $W_G, W(c.v_i), W(d_{ij}), t(c.v_i)$ to 0
3: Set all $W(c)$ to $W_0(c)$
4: $queue \leftarrow root$
5: **while** *queue* is not empty **do**
6: $c \leftarrow queue.poll()$
7: $coeff \leftarrow CalcCoefficient(conditions, c)$
8: $W_G \leftarrow W_G + coeff * W(c)$
9: **for all** $c.v_i$ **do**
10: $W(c.v_i) \leftarrow coeff * W(c) + t(c.v_i)$
11: **if** $c.v_i = D_i$ **then**
12: **for all** $ch \leftarrow \psi(c)$ s.t. $ch.v_i \neq D_i$ **do**
13: $t(ch.v_i) \leftarrow t(ch.v_i) + W_0(c.v_i) * |ch.v_i|/|D_i|$
14: **end for**
15: **end if**
16: **end for**
17: Mark c as calculated
18: **for all** $child \in \psi(c)$ **do**
19: **if** *child* satisfies *conditions* **then**
20: $W(child) \leftarrow W(child) + coeff * W_0(c)$
21: **if** all *child.parents* are calculated **then**
22: $queue.add(child)$
23: **end if**
24: **end if**
25: **end for**
26: **end while**
27: **return** $prob(d_{ij}) \leftarrow W(d_{ij})/W_G$ for all d_{ij}

Algorithm 4 calculates the probabilities in the CCD as described in Section 5.3.

The important part is line 7. *conditions* is a context that contains a conditional situation, if we want to calculate a conditional probability. For finding unconditional probabilities, we put to *conditions* the full domain context, similar to *root*. When we want to obtain probabilities in case that a certain variable has a certain value $v_c = d_c$, we produce a context *conditions* in such a way, that we only allow one value for v_c, but all values for other variables: $conditions = \{v_c = d_c; \forall v_i \in v \backslash v_c : v_i = D_i\}$. This can be combined, to create more sophisticated conditions.

$coeff$ contains a share of a context c that is contained in *conditions*. Later a weight of c and all its variables is reduced to this percent.

Lines 11-15 check the applicability of the $\star$ value to a variable. The context transfers its weight to its children in that case (line 13).

Lines 18-25 go through all children of a *context* and increase their weight. In case all parents are calculated, they are also added to the queue. Each context of the CCD will be queued exactly once, but only in a case it satisfies *conditions* (otherwise its conditional weight is 0).

5.4.1 CCD complexity

Explicit description of different interpretations in a CCD can potentially grow in space exponentially with the size of the environment. However, there are several considerations that help to keep the size of a CCD reasonable.

The biggest growth of a CCD results from faulty contexts. While correct contexts tend to have the same descendants, faulty contexts will generate many new CCD nodes. With the growth of a CCD, one may discard contexts that support the most unlikely interpretations, as most probably they represent faulty or imprecise sensors.

Each environment in a CCD should only contain interdependent variables (i.e. associated by dependency rules, as in Table 5.2). We split independent variables into non-intersecting subgroups and produce a smaller CCD for each subgroup.

Finally, in the next chapter we introduce Reduced Context Consistency Diagrams (RCCD). Reduced CCD remove the need to generate most of the intermediate nodes, keeping the size of a CCD within tractable bounds. The trade off is the decreased querying power.

5.5 Evaluation

The CCD was tested in a living lab constructed on the premises of the University of Groningen. In this section, we describe the living lab setup, the implementation of the CCD component, and the final results.

5.5.1 Living Lab Description

As the setup has an important influence on the rule set of CCD and activity reconginition ontology, we present here the description of our living lab. In our prototype implementation, we make a study in our own offices at the University of Groningen as the living lab. The test site consists of two working rooms that are occupied by two PhD candidates and one coffee corner. The layout of the three-room test site is illustrated together with the ZigBee mesh network of electricity measuring plugs attached to electrical appliances and multi-hop network of simple sensors in Figure 5.5. In particular, we use electricity measuring plugs to detect the power state of five available devices (two PCs, two PC monitors inside two private offices, and a microwave at the coffee corner). Sensors are used to gather other crucial information, pressure sensor for chair's occupancy, acoustic sensor for human voice, and PIR sensor for motion detection.

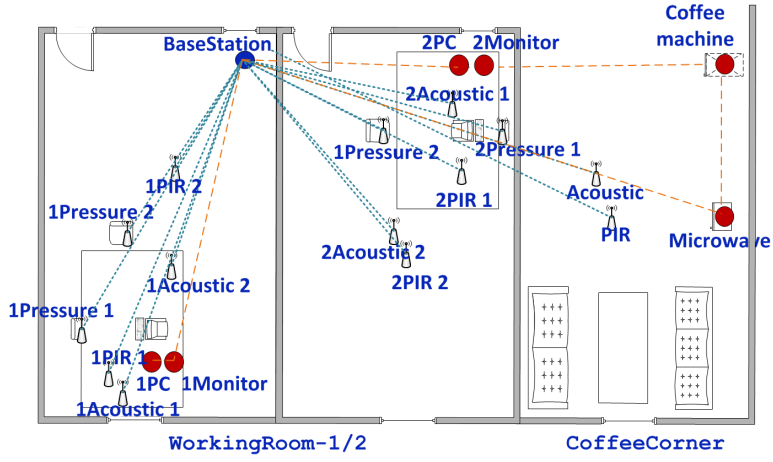


Figure 5.5: Simplified scheme of the living lab setup

5.5.2 CCD implementation

The CCD component contains two main parts: the Web User Interface, which can be used by users of the building to control the rules and variables, and the CCD Manager, which actually creates and manages CCDs, gets sensor readings from the RabbitMQ[1] and sends the results of correction to the Activity recognition component.

The Web UI is quite straightforward. There are three main parts of the component that must be controllable/accessible by users. The first one is the variables configuration of the system. The user can access to see, which variables are currently present in the system, can add/remove variables, and can configure their information, such as the location of a device, the sensor weight, the available states, etc.

The second part that the user can control is the set of rules for the system. The rules are entered as formulas in predicate logic, where every atomic predicate is in a form $v_i = d_{ij}$ or $v_i \in \{d_{ij_1}, d_{ij_2}\}$, and takes $true$ if the variable v_i currently has one of the specified values, and $false$ otherwise. The user has the ability to alter the existing rules, add new ones, or remove the obsolete ones.

Finally, the last part of the Web User Interface is the environment information. The most probable interpretation gets automatically calculated after every sensor readings update, it is always kept in the system and is available for automatic queries or to be shown to a user. In a sense, this is the same information that is presented to the Activity Recognition component, but shown in a human-readable way.

The CCD Manager component manages all system's CCDs and is a common entry point of all input and output information. Potentially, every system can have more than one Context Consistency Diagram. As the diagram only helps to improve information about interdependent variables, there should be a single CCD for every subset of interdependent variables. The variables are interdependent if there are rules which restrict certain values combinations of these variables, or if there is another variable, with which both variables are dependent. For example, variables for PC and monitor are dependent, because the monitor cannot be on if the monitor is off. And two different chairs in the same room are dependent, because if somebody is working with PC, at least one of the two chairs should be occupied, thus there is another variable that combines the two chairs. But a chair in one room is independent from the chair in another room, as there are no rules, no common variables that combine them.

[1] `http://www.rabbitmq.com/`

For the full set of variables the CCD Manager first identifies the subsets of independent variables, and the corresponding rules. Then the CCD Manager creates a CCD structure for every such subset of variables. When a new variable or a new rule is added or removed via user interface, the affected CCD is reconstructed.

The CCD Manager registers itself with the RabbitMQ server, and subscribes for the events of sensors, which are represented as variables. When the CCD Manager receives a new sensor reading, it finds the appropriate CCD based on the variable and sends the reading to this CCD for addition. When the lifetime of a sensor reading expires, the CCD Manager contacts the CCD in order to remove the reading.

The CCD Manager provides a REST interface as well for the activity recognition component (or any other component which may be interested in corrected sensor data). When contacted, it generates a message with a JSON[2] object that contains the current most probable interpretation.

5.5.3 Environment model

The full environment, as modelled in CCD, contains 19 variables. Since not all sensors are equally trustworthy, we introduced initial weights for all sensors, presented in Table 5.4. For example, the Plugwise are the most trustworthy ones, and they have the biggest relative weight, while there are relatively many situations when PIRs fail to detect movement or catch occasional reflection of the sun in an otherwise still environment, so the PIR sensors have the lowest weight. The pressure and acoustic sensors fall inbetween. The weights were assigned based on the empirical evaluation of sensors and several smaller scale experiments to evaluate their accurance rate. Even though in this setting the weights range in the region 0.5-1.0, this does not correspond to the actual probability of the sensor sending a correct value. I.e. sensors with the weight 1.0 still occasionally return erroneous values, while sensors with the weight 0.5 give correct results much more frequently than 50% of the time. The weight of sensors only matters on relative scale, i.e. w.r.t. other sensors, and the absolute value of the sensor weight is not important.

We established eleven rules about the environment for the CCD to construct expanded contexts. For every office 5 rules were designed, and one more rule for the coffee corner. We will now describe all of them.

The first rule was already mentioned as example of sensor dependencies. It is not possible for the monitor to be on if the computer is turned off, therefore the rules are:

[2] `http://json.org/`

Table 5.4: Sensor weights in the CCD

Variable	Weight	Variable	Weight
1AcousticKeyboard	0.8	2LCD	1.0
1Acoustic	0.7	2PC	0.9
1LCD	1.0	2PIRKeyboard	0.5
1PC	0.9	2PIRMotion	0.5
1PIRKeyboard	0.5	2Pressure1	0.8
1PIRMotion	0.5	2Pressure2	0.8
1Pressure1	0.8	3Acoustic	0.7
1Pressure2	0.8	3Microwave	1.0
2Acoustic	0.7	3PIRMotion	0.5
2AcousticKeyboard	0.8		

$1LCD = true \Rightarrow 1PC = true$
$2LCD = true \Rightarrow 2PC = true$

The monitor was configured to turn off after 1 minute of inactivity, thus we could ensure that it is running only when people are actually working with the computer. A working person should be sitting on the chair and the PIR sensor that catches only the keyboard and mouse area should actually catch the respective activity. If the PIR Keyboard detects a movement of the mouse or above the keyboard, someone must be sitting and moving the mouse or typing.

$$1LCD = true \Rightarrow (1Pressure1 = true \vee 1Pressure2 = true) \wedge 1PIRKeyboard = true$$

$$2LCD = true \Rightarrow (2Pressure1 = true \vee 2Pressure2 = true) \wedge 2PIRKeyboard = true$$

$$1PIRKeyboard = true \Rightarrow (1Pressure1 = true \vee 1Pressure2 = true)$$

$$2PIRKeyboard = true \Rightarrow (2Pressure1 = true \vee 2Pressure2 = true)$$

The Acoustic Keyboard sensor can be easily enabled on the same device that contains the PIR Keyboard sensor. Thus, we have an additional source of information. Therefore, the Acoustic Keyboard sensor acts as a backup for the main acoustic sensor in the room:

$$1AcousticKeyboard = true \Rightarrow 1Acoustic = true$$

$$2AcousticKeyboard = true \Rightarrow 2Acoustic = true$$

During our experiment the acoustic sensors were configured to catch human voices. We also ensured the working atmosphere in the offices, i.e. no music running in the background when no one is around. Given the setting, the recongnition of a sound inside a room meant people speaking inside, which gives enough movement for the PIR sensors to recognize it. To ensure this the following rules were added:

$$1Acoustic = true \Rightarrow 1PIRMotion = true$$

$$2Acoustic = true \Rightarrow 2PIRMotion = true$$

When the meeting is taking place, conversations are expected, which should be caught by the acoustic sensor:

$$1Pressure1 = true \wedge 1Pressure2 = true \Rightarrow 1Acoustic = true$$

$$2Pressure1 = true \wedge 2Pressure2 = true \Rightarrow 2Acoustic = true$$

Finally, for the coffee corner, if people are around and either speaking or using the microwave, we expect the PIR sensor to detect them:

$$3Acoustic = true \vee 3Microwave = true \Rightarrow 3PIRMotion = true$$

The CCD is constructed based on those rules, and every sensor reading increases the probability only of those situations that are consistent with these rules and this reading. The weight of the sensor is added to these situations (CCD leaves) with every reading and removed when the reading becomes obsolete. When contacted by AR component, the situation with the biggest weight is returned.

5.5.4 Results

We performed an experiment in three separate days in a week, that is, daily from 10:00 to 17:00, on March 26, March 29, and April 3, 2013 to verify the accuracy of the proposed architecture and approach in terms of activity classification, but also the error correction rate of CCD. The experiments concern the recognition of six different activities performed by two PhD candidates in `WorkingRoom-1`, `WorkingRoom-2`, and `CoffeeCorner`. During the experiment, the users take accurate notes of actual activities happening in the offices every minute, which is used as golden truth for the evaluation. Table 5.5 summarizes the actual occurrences of activity instance at each activity area, that is then used as ground truth for evaluation. These ground truth occurences were collected by PhD students that

Table 5.5: Ground truth of the occurrences of activity instances

Area	Activity					
	A	B	W	Wpc	M	C
`WorkingRoom-1`	70	232	256	640	3	0
`WorkingRoom-2`	129	112	139	754	67	0
`CoffeeCorner`	843	0	0	0	0	358

A = Absence; B = Being present; W = Working without PC; Wpc = Working with PC;M = Having a meeting/discussion; C = Coffee break

were working in these rooms by recording manually all activities they performed thoughout the days of the experiment. By using 1 minute time interval, the collected ground truth is composed of 1201 activity instances.

In order to evaluate how CCD is able to detect and correct the errors in sensor readings and how the CCD corrections affect the accuracy of our recognition solution, we implement another system which uses the same architecture except the CCD-based fault correction component. Instead, to correct the faults in sensor readings, we implement a simple major voting algorithm, that is, the value of a sensor reading (TRUE or FALSE) is decided based on the most probable observation for a given combination of five readings collected in a minute. We call our algorithm *Averaged* in order to distinguish it from a CCD solution.

The Gateway provides raw sensor readings for both CCD and Averaged components. After processing to correct the errors, CCD and Averaged provide corrected sensor readings as inputs for Activity Recognition component. The recognition results by using CCD and Averaged sensor readings are stored to make comparisons.

The overall success rates of the system for each monitored area are shown in Table 5.6. The first impression is that CCD significantly helps to correct the faults in sensor readings, thus the success rates notably increase, compared to Averaged ones. In particular, CCD helps to correct 90 minutes of wrong recognition at `WorkingRoom-1`, equivalent to 40%, improving the success rate for this room by 7.66%. The CCD corrections for `WorkingRoom-2` are even more significant with 46.83% of faults corrected, thus the accuracy of the recognition reaches 87.42%. That means it is 11.07% better compared to 76.35% returned by Averaged correction. One witness the most remarkable corrections at `CoffeeCorner`, where 82.60% of faults are corrected by CCD component. The high number of corrections can

be explained by investigating the raw readings from sensors at the `CoffeeCorner`. Because CCD helps to correct PIR sensor readings with the following rule:

$$3Acoustic = true \vee 3Microwave = true \Rightarrow 3PIRMotion = true$$

Table 5.6: Success rates at each area

Area	$\sum_{\forall activities} FN$		**% of errors**
	Averaged	**CCD**	**fixed**
`WorkingRoom-1`	230	138	40.00%
`WorkingRoom-2`	284	151	46.83%
`CoffeeCorner`	46	8	82.60%
$T_{all} = 1201$ *minutes*			
Area	**Success rates**		
	Averaged	**CCD**	**Improvement**
`WorkingRoom-1`	80.85%	88.51%	7.66%
`WorkingRoom-2`	76.35%	87.42%	11.07%
`CoffeeCorner`	99.17%	99.33%	00.16%

The improvement of success rates is also reflected in the Figure 5.6, which shows comparison between actually happened activities in each monitored area and the recognized activities from Averaged result and CCD result.

Table 5.7 to Table 5.9 illustrate the result in more detail by showing recall and precision of all activities and all areas. Recall of an activity is the ratio of correctly identified activity occurences to all actual occurences of this activity. Precision is the ratio of correctly identified activity occurences to the number of all occurences that were identified as this activity. In all three areas, CCD improves recall of *Working with PC*, *Having a meeting/discussion*, and *Having a coffee break* activities, making sure that PC/meeting-related devices are working while the users are really working with the PC or having a meeting. These satisfy one of the important criteria of a smart building that is the user comfort has higher priority than energy saving. For example, recall of *Working with PC* at `WorkingRoom-1` improves from 94.53% to 99.06%, while this improvement at `WorkingRoom-2` is from 81.69% to 97.61%.

During the experiments, we also measure the running time of our system. The results show that our current system, which includes three activity areas (two

Table 5.7: Results for working room 1

(a) Confusion matrix

Recognized as →	Wpc		W		M		B		A		U	
	Aver.	CCD	Aver.	CCD	Aver.	CCD	Aver.	CCD	Aver.	CCD	Aver.	CCD
Working with PC	**605**	**634**	8	2	0	0	2	4	0	0	25	0
Working without PC	1	9	**255**	**247**	0	0	0	0	0	0	0	0
Having meeting/discussion	1	1	2	2	**0**	**0**	0	0	0	0	0	0
Being present	1	6	12	17	0	0	**46**	**117**	168	92	5	0
Absence	0	0	3	5	2	3	0	0	**65**	**62**	0	0

(b) Precision/Recall

Activity	Precision (%)		Recall(%)	
	Aver.	CCD	Aver.	CCD
Working with PC	99.50	97.54	94.53	99.06
Working without PC	91.07	90.47	99.60	96.48
Having a meeting/discussion	0	0	0	0
Being present	95.83	96.69	19.82	50.43
Absence	27.89	40.25	92.85	88.57

Wpc = Working with PC; W = Working without PC; M = Having a meeting/discussion;B = Being present; A = Absence; U = Unrecognised minutes

Table 5.8: Results for working room 2

(a) Confusion matrix

Recognized as →	Wpc		W		M		B		A		U	
	Aver.	CCD	Aver.	CCD	Aver.	CCD	Aver.	CCD	Aver.	CCD	Aver.	CCD
Working with PC	**616**	**736**	14	14	0	0	3	4	1	0	120	0
Working without PC	4	4	**130**	**131**	0	0	3	2	2	2	0	0
Having meeting/discussion	12	37	4	5	**0**	**25**	0	0	0	0	51	0
Being present	0	2	6	17	0	0	**44**	**32**	62	61	0	0
Absence	0	2	0	0	0	0	1	1	**127**	**126**	1	0

(b) Precision/Recall

Activity	Precision (%)		Recall(%)	
	Aver.	CCD	Aver.	CCD
Working with PC	97.16	94.23	81.69	97.61
Working without PC	84.41	78.44	93.52	94.24
Having a meeting/discussion	0	0	-	37.31
Being present	86.27	76.92	39.28	26.78
Absence	66.14	66.31	98.44	97.67

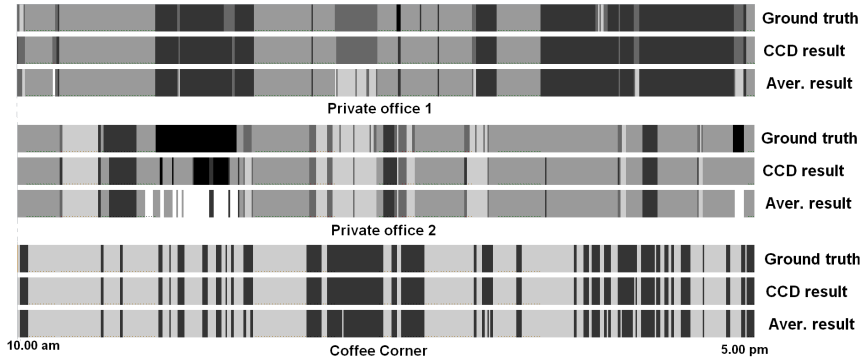
Wpc = Working with PC; W = Working without PC; M = Having a meeting/discussion;B = Being present; A = Absence; U = Unrecognised minutes

Table 5.9: Results for coffee corner

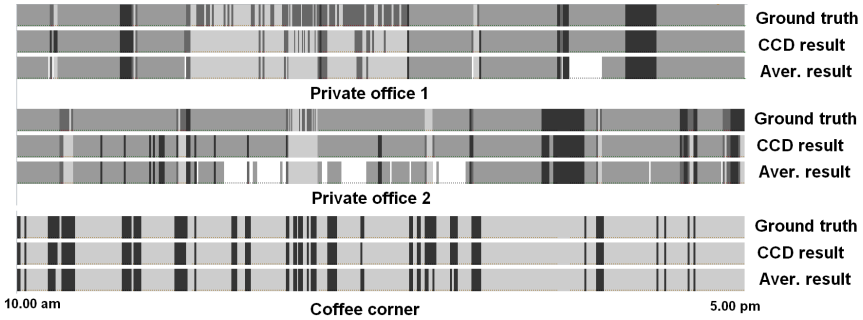
(a) Confusion matrix				
Recognized as →	C		A	
	Aver.	CCD	Aver.	CCD
Having a coffee break	**312**	**350**	46	8
Absence	0	0	**843**	**843**
(b) Precision/Recall				
Activity	Precision (%)		Recall(%)	
	Aver.	CCD	Aver.	CCD
Having a coffee break	100	100	87.15	97.76
Absence	94.82	99.06	100	100

C = Having coffee break; A = Absence

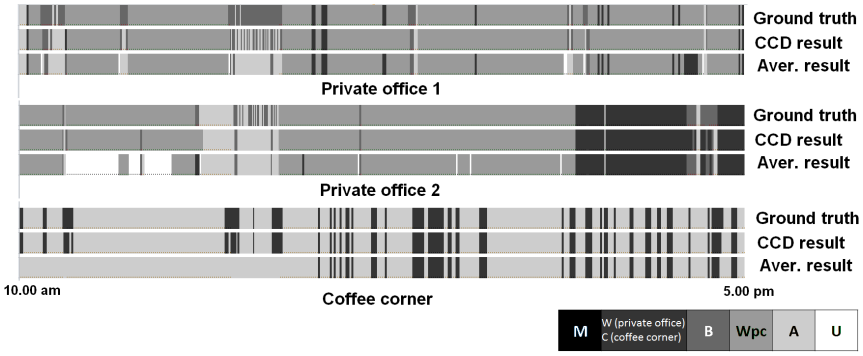
working rooms and a coffee corner) and 19 sensors (15 simple sensors and five electricity measuring plugs), can run in a distributed manner with the running time of less than 5ms.



(a) Result day 1



(b) Result day 2



(c) Result day 3

Figure 5.6: Experiment in three days

Chapter 6

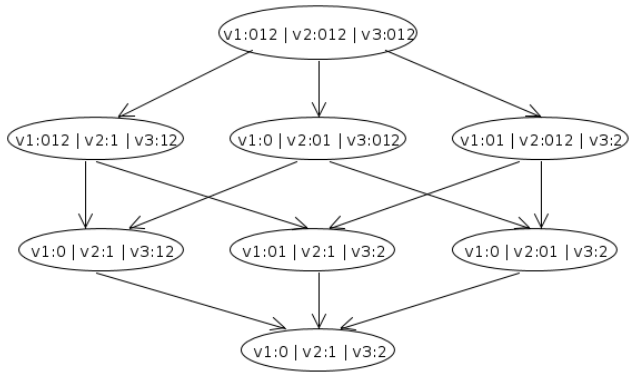
Reduced Context Consistency Diagrams for Resolving Data Inconsistencies

Context consistency diagrams (CCD) are capable of storing sensor data and defining several possible context interpretations in the presence of a conflict or an ambiguity, either because of incomplete knowledge about the environment, or because of erroneous sensor readings. If no information is discarded, further sensor readings help to refine the knowledge and make more informed decisions about the correctness of certain sensor readings. In this chapter, we extend the notion of a context consistency diagram and introduce a reduced context consistency diagram (RCCD) for dealing with inconsistent and incomplete data, which severely reduces the resulting diagram size comparing to the original full CCD. This is done by removing intermediate nodes with no initial weight during the diagram generation, and the trade off is the reduced querying capabilities of the diagrams. Both full and reduced CCDs are capable of storing all the information without discarding anything, even if the data has conflicts. RCCD, while having a less extensive querying capabilities than CCD, requires much less computational and storage power. Pervasive systems can implement either CCD or RCCD mechanism to deal with ambiguous or conflicting context data.

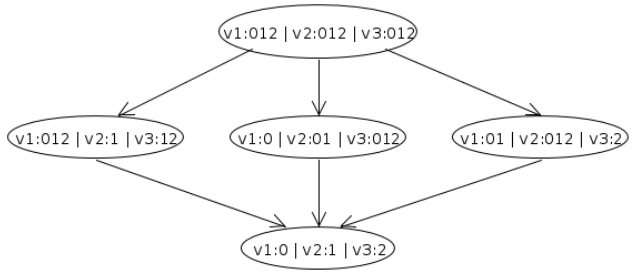
6.1 Reduced context consistency diagram

While CCD provides a full picture of the information together with existing inconsistencies, it is an extensive structure that shows all the possibilities of environment knowledge explicitly, thus at the expence of computational and storage power.

For this we devise a way to reduce a CCD while still keeping the knowledge about existing inconsistencies intact. The reduced context consistency diagram (RCCD) uses the fact that some of those nodes that are not created directly from sensor readings, but are created as intermediate ones, are combined together to create a more knowledgeable common descendant. If this is the case, those intermediate nodes can be truncated from the diagram, while still keeping the information



(a) Full CCD of three contexts



(b) Reduced CCD of the same contexts

Figure 6.1: Example of a full CCD (a) and a corresponding reduced CCD (b)

about the most probable situation.

For example, in the case described in Figure 6.1a, a full CCD is shown for the following three contexts:

$$c_1 = (v_1 \in (0,1,2);\ v_2 = 1;\ v_3 \in (1,2))$$

$$c_2 = (v_1 = 0;\ v_2 \in (0,1);\ v_3 \in (0,1,2))$$

$$c_3 = (v_1 \in (0,1);\ v_2 \in (0,1,2);\ v_3 = 2)$$

However, the three generated nodes of the second tier can be reduced, as they all are combined in the more knowledgeable context ($v_1 = 0; \quad v_2 = 1; \quad v_3 = 2$). The corresponding reduced CCD is shown in Figure 6.1b.

Notice that RCCD can reduce only nodes that were not originally obtained from sensor readings (in other words, those nodes that do not belong to C_0 group and do not have associated initial weight $w_0(c)$). In the same example, the full CCD will be exactly the same as reduced CCD (both shown in Figure 6.1a) in case we also receive three more sensor readings that account for the following contexts:

$$c_4 = (v_1 = 0;\ v_2 = 1;\ v_3 \in (1, 2))$$

$$c_5 = (v_1 \in (0, 1);\ v_2 = 1;\ v_3 = 2)$$

$$c_6 = (v_1 = 0;\ v_2 \in (0, 1);\ v_3 = 2)$$

The formal definition of RCCD goes as follows:

Definition 8 (Reduced context consistency diagram (RCCD)). Given an environment $\langle V, D\rangle$ and a set of contexts $C_0 = \{c_k\},\ k \in 1..N$, a *reduced context consistency diagram (RCCD)* is a tuple $G_r = \langle C, E, r\rangle$, where:

- $r = D$, is a special context, the *root*;
- $C = C_0 \cup C_r \cup r$ where C_r is defined as $C_r = \bigcup c_p \in C_u \backslash (C_0 \cup r)$ *s.t.* $\nexists c_c \in C_u \cup C_0 :\ (c_c \subset c_p))$
- $E \subseteq C \times C$, such that $(c_2, c_1) \in E$ iff $\exists c_1, c_2 \in C : c_1 \subset c_2$ and $\nexists c_m \in C : c_1 \subset c_m \subset c_2$.

The only difference with the Definition 7 of CCD is the definition of a context set C, which, however, drastically changes the resulting diagram.

To describe properties of the RCCD we split the contexts of the CCD into the two categories: solid and non-solid. A set of **solid** contexts C_s consists of the root and all contexts out of the original context set C_0. **Non-solid** contexts C_{ns} are all the contexts out of a full set of intersection of a power set C_u that are not included into a solid set (thus that are not among original contexts).

$$C_s = C_0 \cup r \tag{6.1}$$

$$C_{ns} = C \backslash C_s \tag{6.2}$$

As in CCD, vertices of the RCCD include all the solid contexts. However, unlike in CCD, where all non-solid contexts are also kept, in RCCD they are kept in the diagram only in case they are the most knowledgeable contexts.

This leads us to the first property of the RCCD:

Property 4. All non-solid nodes of RCCD do not have children.

Proof. If a node $c \in C$ has a child, then by the definition of a context set C, this node may not be the part of C_r, because $\forall c \in C_r : \nexists c_c \in C_u \cup C_0 : \ (c_c \subset c)$. From facts that $C = C_0 \cup C_r \cup r$, and $c \in C$, but $c \notin C_r$, follows that $c \in C_0 \cup r$, and by definition of a solid node from equation 6.1, c must be solid. Thus if a node of RCCD has a child, it is always a solid node. Thus non-solid nodes do not have children. □

The second property of RCCD is:

Property 5. If all nodes with children of a full CCD are part of the original context set C_0, then a corresponding reduced CCD is equal to the full CCD.

Proof. Solid nodes $C_0 \cup r$ are always part of both CCD and RCCD context sets, thus only non-solid nodes differ. We are given that all nodes of CCD with children are in C_0 set, thus for the remaining nodes the following holds:

$$\forall c \in C_u \backslash (C_0 \cup r) : \nexists c_c \in C_u \cup C_0 : \ (c_c \subset c))$$

which is exactly the definition of corresponding C_r from RCCD. So

$$\forall c \in C_u \backslash (C_0 \cup r) : c \in C_r$$

or

$$C_u \backslash (C_0 \cup r) = C_r$$

Thus,

$$C^{full} = C_u \cup C_0 \cup r = (C_u \backslash (C_0 \cup r)) \cup C_0 \cup r = \\ = C_r \cup C_0 \cup r = C^{reduced}$$

The fact that $C^{full} = C^{reduced}$ also means that $E^{full} = E^{reduced}$ by definition of edges construction. So CCD and RCCD under the given conditions are equal. □

All the original properties of CCD also hold for RCCD. Those are:

Property 6. For a given set of contexts there is one and only one non-isomorphic representation of its RCCD.

Property 7. The order of contexts addition does not change the resulting RCCD.

Property 8. Adding and then removing a context does not change the resulting RCCD.

The proof of these three properties for RCCD is exactly the same as for CCD and is presented in Section 5.4.

6.2 RCCD maintenance

In this section we present algorithms to add and remove a context to the RCCD.

The addition of a new context to the RCCD is shown in Algorithm 5. The algorithm is split into two parts: *AddContext* recursively searches predecessors of the *context* to find its parents, i.e. contexts, to which the new context should be added as a child. As soon as those parents are found, the second part, *CheckChildren*, recursively checks if a *context* is consistent with its siblings, and a child should be generated.

The first part, *AddContext*, checks if a *parent* is a parent of a *context*. It starts by checking if there is a child of a *parent* equal to the *context*. If this is the case, a corresponding weight is added to the initial weight of a child and the algorithm ends.

Otherwise it checks if there are any non-solid children of a *parent* that are more general than the *context*. In this case, all the parents of such children are moved and become parents of the *context*. Note that this effectively deletes those children from the diagram, because they do not have parents anymore, and non-solid nodes never have their own children.

If no suitable children are found yet, the algorithm checks if there are solid children of a parent that are more general than the *context*. In case they are found, the algorithm repeats recursively for them.

Otherwise we found a parent of the *context*. The second part of the algorithm, *CheckChildren*, is then called and the *parent* adds the *context* as a child.

The *CheckChildren* function receives two nodes as input, a *context*, which is a newly added to the diagram node, and another *node*, for which we suspect that its children may be consistent with a *context*. We process all the children of the *node* in the following manner:

If a *child* is included into a *context*, we add it as a descendant to a *context* and remove it as a child from *node*, in case a *context* is already a child of a *node*.

Otherwise we check for consistency of a *context* and a *child*. If they are consistent and a *child* is non-solid, we remove it from the diagram, and put an intersection of a *child* and *context* in its place. We also add a link from a *context* to the new node.

If a child is solid, however, we check if there is already (or should be created) a common descendant between a *child* and a *context* by calling *CheckChildren* recursively on those nodes. If it returns negative result, we add a new node (their intersection) to the diagram by adding it as a child to a *context* and a *child*.

Algorithm 6 describes the removal of an outdated context from RCCD.

Algorithm 5 Adding context to RCCD

```
1: function AddContext(context, parent, weight)
2: if ∃ch ∈ parent.children s.t. ch = context then
3:    W_0(ch) ← W_0(ch) + weight
4:    exit
5: else if ∃child ∈ parent.children.nonsolid s.t. context ⊂ child then
6:    ∀child :Move parents from child to context
7: else if ∃child ∈ parent.children.solid s.t. context ⊂ child then
8:    ∀child :AddContext(context, child, weight)
9: else
10:   CheckChildren(context, parent)
11:   Add link from parent to context
12: end if
13:
14: function CheckChildren(context, node):boolean
15: result ← false
16: for all child ∈ node.children do
17:   if child ⊂ context then
18:     if node is a parent of context then
19:       Remove link from node to child
20:     end if
21:     Add child to context as descendant
22:     result ← true
23:   else if isConsistent(context, child) then
24:     x ← context ∩ child
25:     if ¬isSolid(child) then
26:       Move parents from child to x
27:       Add link from context to x
28:     else if ¬CheckChildren(context, child) then
29:       Add link from context to x
30:       Add link from child to x
31:     end if
32:     result ← true
33:   end if
34: end for
35: return result
```

Algorithm 6 Removing context from RCCD

```
1: function RemoveContext(context, weight)
2: W_0(context) = W_0(context) − weight
3: if W_0(context) = 0 then
4:    if context has children then
5:       for all parent ∈ context.parents do
6:          Remove link from parent to context
7:          Add all context.children to parent as descendants
8:       end for
9:       nodes ← context.children.nonsolid
10:      Remove links from context to all its children
11:   else
12:      nodes ← context
13:   end if
14:   for all node ← nodes do
15:      x ← ⋂ node.parents
16:      if x ≠ node & ∄brth ∈ node.brothers s.t. brth ⊂ x then
17:         Add link from all node.parents to x
18:         Remove links from all node.parents to node
19:      end if
20:   end for
21: end if
```

First of all, the context can only be removed in case the context becomes non-solid after reducing the initial weight by the amount, corresponding to the outdated sensor reading. I.e. in case there are no more other sensor readings that support this context.

If a *context* has children, we first remove a link from context parents to it, and then add links from context parents to its children, connecting them directly.

After this, on lines 14-20 we check non-solid children of a *context*, or a context itself in the absence of children, for the maximum generality. That means that each such *node* should stay on the diagram only if it still has more than one parent, and if the intersection of all its current parents is exactly equal to a *node*. It may be the case that after the removal of a parent, a non-solid *node* is now less general than it should be. In this case we create a new node x, which is the intersection of all its current parents, and move all the parents of a *node* to x, effectively removing a *node* from the diagram.

6.3 RCCD reasoning

In Section 5.4 we describe a way to calculate probabilities of all the interpretations possible using CCD. The method is applicable and very useful in situations where sensor readings are highly erroneous, so the system must account for interpretations that are second- or third- most probable. The RCCD can not be used for these kinds of queries, because second-, third-, etc. most probable interpretations are often reduced in favor of the single most probable one. However, RCCD is capable of answering to the question "What is the most probable situation at this moment?". To prove this, we use the fact that the most probable situation is always the one among the most knowledgeable situations, i.e. those that do not have children. Indeed, let's assume that a context that contains the most probable situation has a child. A child is always more knowledgeable than a parent, in other words it contains a subset of interpretations that a parent contains. As a child has its own weight, only those interpretations of a parent that are also contained in a child gain this additional weight. Which by itself means that they become more probable, than the others. So, we proved that interpretations that are contained in a child will always be more probable than interpretations of a parent, which are not contained in any of its children. Which means that the most probable interpretation of a situation is always the one among nodes with no children.

RCCD by Definition 8 never reduces nodes that do not have children, so they are always present on a diagram. Moreover, all the initial weights from original contexts are also transferred to these nodes, and all the consistency among original nodes is kept, as the way of inheriting nodes in RCCD is the same, as in CCD. Due to these facts the most probable interpretation has the biggest weight among all interpretations. The probability calculation, presented in Section 5.4, still can be used in RCCD for finding the most probable interpretation.

As well as CCD, RCCD never discards any information from sensors, and if the most probable interpretation changes with the arrival of a new context, RCCD immediately catches this change.

6.3.1 Unfolding of RCCD to CCD

The previous subsection shows that for systems that are mostly concerned with the question "What is the most probable situation at this moment in time?" RCCD is a more preferable choice of a diagram than CCD. However, even for such systems sometimes there are cases when additional information about other possible situations or conditional probabilities is important. Fortunately, the choice of RCCD over CCD does not permanently hinder the ability to obtain the answers to these

queries. We present Algorithm 7 that allows to unfold a RCCD to obtain a full CCD.

The function $UnfoldNode$ will be called for every node of the diagram. At the beginning of the algorithm the *queue* contains all the most knowledgeable nodes (described by $RCCD.lastnodes$), i.e. nodes that do not have children. In other words, the algorithm unfolds nodes from the bottom to the top. The last node to be unfolded is always the *root*.

$UnfoldNode$ is called either for all or for a subset of node parents. When a node is polled from a *queue*, the function is always called for all node parents, but later it can be recursively called for a subset of them. The function checks if there is only one parent among the input *parents*. If it is the case, it checks if all the children of a parent are already marked, and if it is the case, it adds a *parent* to the *queue*. If there are more than one parent in *parents* subset, the function tries to remove a single parent from this subset one by one, and checks, if the remaining parents can create a more general consistent child x, than a *node*. If it is the case, the link from all such parents to the *node* is removed, and x is added as a child to them instead. Also each parent checks if it has other children that either should now be the children of x (in this case the link from *parent* to *child* is removed, and a link from x to *child* is added instead), or that have a consistent child y with x (y is than added as descendant to both *child* and x in this case). After this is done, the link from x to *node* is added and a new node x is put to the queue.

6.4 CCD vs RCCD complexity

Explicit description of different interpretations in a CCD can potentially grow in space exponentially with the number of distinct contexts in the original set. However, there are several considerations that help to keep the size of a CCD reasonable.

The biggest growth of a CCD results from faulty contexts. While correct contexts tend to have the same descendants, faulty contexts will generate many new CCD nodes. With a growth of a CCD, one may discard contexts that support the most unlikely interpretations, as most probably they represent faulty or imprecise sensors.

Each environment in a CCD should only contain interdependent variables. We split independent variables into non-intersecting subgroups and produce a smaller CCD for each subgroup.

RCCD, on the other hand, produces a much smaller diagram. First of all, notice that RCCD does not generate new nodes, unless they are the most knowledgeable.

Algorithm 7 Unfold RCCD to CCD

```
1: function UnfoldRCCD
2: queue ← RCCD.lastnodes
3: while queue is not empty do
4:    node ← queue.poll()
5:    UnfoldNode(node, node.parents)
6: end while
7:
8: function UnfoldNode(node,parents)
9: Mark node
10: if parents.size = 1 then
11:    if parents(0).children are marked then
12:       queue.add(parents(0))
13:    else
14:       for all par ∈ parents do
15:          x ← ⋂(parents\par)
16:          if x ≠ node then
17:             for all parent ∈ parents\par do
18:                Remove link from parent to node
19:                for all child ∈ parent.children do
20:                   if child ⊂ x then
21:                      Remove link from parent to child
22:                      Add link from x to child
23:                   else if isConsistent(x, child) then
24:                      y ← x ∩ child
25:                      Add y as descendant to child
26:                      Add link from x to y
27:                   end if
28:                end for
29:                Add link from parent to x
30:             end for
31:             Add link from x to node
32:             queue.add(x)
33:          else
34:             UnfoldNode(node,parents\par))
35:          end if
36:       end for
37:    end if
38: end if
```

In case all the contexts are correct, the maximum number of nodes in RCCD will be $N_c + 2$, where N_c is the number of distinct nodes in the original context set, and the number 2 corresponds to the *root* and a possibly generated single child. The child is single, because if all contexts are correct, they are all consistent with each other, thus they all have the same most knowledgeable descendant.

Each erroneous context potentially adds N_v new children (alternative "most knowledgeable" nodes), one per each variable, where N_v is the number of variables. Thus with the presence of erroneous contexts the maximum number of nodes in RCCD is equal to $N_c + N_v * N_{err} + 2$ where N_{err} is the number of erroneous contexts. Notice that normally we assume a situation, where $N_{err} \ll N_c$, thus we expect small sizes of RCCD in practice. If this is not the case, with bigger numbers of errors the RCCD size will grow as well.

In table 6.1 we compile all the previously given information to present a concise comparison between the two structures. That should help to decide, which of the structures is better suited for given projects.

6.5 Evaluation

Our evaluation section is split into two parts. We start by showing a sample run of the system and discuss it in details. Then we make a general overview of system's performance based on several experiments, and study the dependence of system's performance on several system parameters.

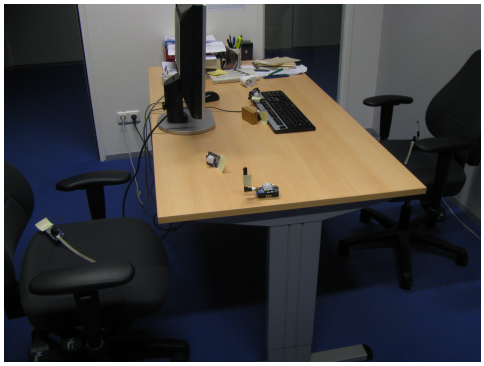
To evaluate the system in real conditions, we performed an experiment with several sensors. The setup of the experiment can be seen in Figure 6.2.

We used six sensors altogether: 2 acoustic sensors, 2 PIR motion sensors, and 2 pressure sensors. The sensors are produced by Advantic Systems[1]. They are IEEE 802.15.4 compliant wireless sensors that use open-source "TelosB" platform [Nguyen and Aiello, 2012]. All sensors are equipped with ultra low-power 16bit microcontroller MSP430. The pressure sensor uses the Tekscan® A201-100 FlexiForce® sensor, which provides force and load measurements for both static and dynamic forces (up to 100lb or 400N). The Passive InfraRed (PIR) motion sensor uses the Perkin Elmer Optoelectronics® LHI878 sensor to detect motion in the given direction. The SE1000 acoustic sensor has a mini-microphone (20-16000 Hz, SNR 58 dB) that is designed to detect the presence of sound. All sensors were configured to measure intensity of corresponding signals and thresholds were applied to readouts of each sensor in order to return a higher-level boolean value (presence or absence of sound/motion/pressure) once every second. The data was

[1] `http://www.advanticsys.com`

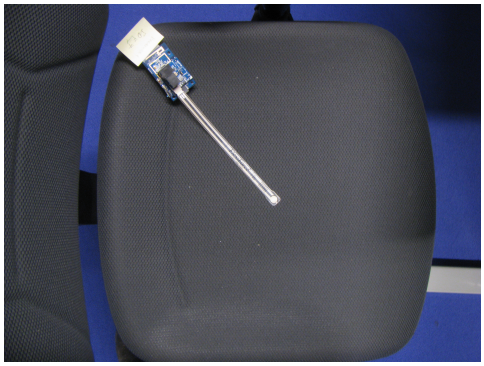
	CCD	RCCD
Computation effort	Is rather computationally heavy, though ways exist to keep the CCD within a given size, while losing some information.	Computationally light, can be maintained and updated much faster than CCD, and is capable of handling more data without losing information.
Information handling	Keeps all the arrived information, and handles inconsistencies with the existing information. Further information updates can change the most probable situation.	
Quering and reasoning	Can be used to find the most probable situation. Also can rank other situations by their probability, and answer to different kinds of queries, such as "What are the situations that are at least 20% probable?", "What is the second, third, etc. most probable situation?", "What is the probability distribution of values of the certain variable?", etc.	Has a simple and fast way to find the most probable situation. Answers to this question much faster than CCD.
Inter-dependence	If needed, RCCD can be unfolded into a full CCD at any moment in time.	

Table 6.1: CCD vs RCCD comparison



(a) Overview on the experiment's location setup

(b) Acoustic and PIR motion sensors



(c) Pressure sensor

Figure 6.2: System's experimental setup

then sent to the RCCD structure, which in turn returned the current most probable situation.

Figure 6.2a shows the general placement of the sensors. The setup shows the office of a single person and the idea of using such sensors is to be able to recognize current activity of a person. Among those activities we assume work with or without PC, meeting with another person, or absence from the working desk. While each of the sensors occasionally produces faulty readings, their mutual dependencies that we described in terms of rules, help RCCD to recognize the correct situation.

First we describe what each sensor is aimed to recognize, then we describe their mutual interdependecies, and afterwards we show the results of the experimental run of such a system.

6.5.1 Sensors description

Pressure sensor 1 (PR_1) is located on a chair of the main person in front of the PC, and is triggered if someone is sitting on this chair.

Pressure sensor 2 (PR_2) is located on a "guest" chair, and is triggered if someone is sitting on it.

Acoustic sensor 1 (AC_1) is placed near the keyboard and is aimed to detect the sound of keys being pressed, in order to recognize the typing activity.

Acoustic sensor 2 (AC_2) is a general acoustic sensor that is aimed to recognize a sound in a room. It is placed in between the two chairs, because the sound usually means the conversation between the two people.

Keyboard typing, while triggering sound detection on AC_1, is not loud enough to trigger sound detection on AC_2, thus AC_2 remains silent in this case. However, when two people are speaking with each other, both acoustic sensors detect sound. So we can only definitely recognize the keyboard typing when AC_2 is silent, while AC_1 is detecting sound.

PIR motion sensor 1 (M_1) is directed exactly at the front of the PC (looking directly at the first chair), and is aimed to detect any motion in this direction. The sensor gives us additional information and can help to recognize inaccuracies of other relevant sensors.

PIR motion sensor 2 (M_2) is looking directly at the guest chair, and is aimed to detect any motion on or around this chair. This sensor as well gives us additional information and can help to recognize inaccuracies of other relevant sensors.

Table 6.2: Experiment results

Sensor	Number of errors			Error rate,%			% of errors fixed		
	Latest	**Aver.**	**RCCD**	**Latest**	**Aver.**	**RCCD**	**Latest**	**Aver.**	**RCCD**
M1	631	563	77	35.06	31.28	4.28	0.0	10.78	87.80
PR1	270	259	274	15.00	14.39	15.22	0.0	4.07	-1.48
AC1	398	336	314	22.11	18.67	17.44	0.0	15.58	21.11
M2	132	116	11	7.33	6.44	0.61	0.0	12.12	91.67
PR2	18	8	9	1.00	0.44	0.50	0.0	55.56	50.00
AC2	160	126	123	8.89	7.00	6.83	0.0	21.25	23.13
Total	1609	1408	808	14.90	13.04	7.48	0.0	12.49	49.78

6.5.2 Interdependencies rules

As already noted, sensors in our case study have common dependencies. For example, when AC_2 is detecting sound, AC_1 is also detecting sound (but not the other way around). When a person is sitting in a chair, both pressure PR_1 and motion M_1 sensors will detect activity. These and other dependencies are captured by creating the following rules:

"Pressure implies motion." Both motion sensors are directed exactly on the chairs, and located very closely to them. When someone is sitting on a chair, in most cases the motion sensors detect small motions of a person in it. We help our system to detect the faulty readings of no motion by adding these two rules:

$$PR_1 \implies M_1 \tag{6.3}$$

$$PR_2 \implies M_2 \tag{6.4}$$

"Who is typing, if no one is there?" If we detect no motion, and no pressure on the chair in front of the PC, and there is no general sound in the room, the keyboard acoustic sensor should also remain silent. Hence the third rule:

$$\neg PR_1 \wedge \neg M_1 \wedge \neg AC_2 \implies \neg AC_1 \tag{6.5}$$

Note that given Rule 6.3, we can cancel out the variable PR_1 from this formula. However, we prefer to keep it in this format both for ostensive purposes and for each rule to remain completely independent from other rules.

"I heard something. Did you hear it?" The first acoustic sensor is placed very close to the keyboard in order to detect soft noise of typing, which second

sensor is unable to detect. The second sensor, however, is placed just in the middle of the meeting area in order to detect all loud noises in the room, the most common noise being human speech. The first sensor is able to detect all those loud noises as well, which means it should always be triggered when the second acoustic sensor detects something:

$$AC_2 \implies AC_1 \tag{6.6}$$

"Room is busy." If we detect sound by the keyboard acoustic sensor, and a motion in general area, but no pressure on the chair in front of the PC, it means the sound comes from somewhere else in the room, so the second acoustic sensor should also be able to detect it.

$$AC_1 \wedge M_2 \wedge \neg M_1 \implies AC_2 \tag{6.7}$$

6.5.3 System's run

We did an experimental run of the system with the abovementioned setup to evaluate to which extent RCCD is able to detect the correct situation. The system was collecting data for 30 minutes, during which the situation was the following:

1. For the first 10 minutes a person was working with the computer. Thus the expected correct values of sensors would be:

   $M_1 = true;\ PR_1 = true;\ AC_1 = true;$
   $M_2 = false;\ PR_2 = false;\ AC_2 = false$

2. Then the short meeting with another person was held for 5 minutes. The expected values are:

   $M_1 = true;\ PR_1 = true;\ AC_1 = true;$
   $M_2 = true;\ PR_2 = true;\ AC_2 = true$

3. After this the person was reading papers silently for 10 minutes. Corresponding expected sensor values:

   $M_1 = true;\ PR_1 = true;\ AC_1 = false;$
   $M_2 = false;\ PR_2 = false;\ AC_2 = false$

4. Last 5 minutes the room was empty:

   $M_1 = false;\ PR_1 = false;\ AC_1 = false;$
   $M_2 = false;\ PR_2 = false;\ AC_2 = false$

6.5.4 Results

The experiment was running for 30 minutes with sensors sending their readings each second. The lifetime of sensor readings was set to 5 seconds, so for each sensor at any moment in time we had 5 latest readings that were to be considered. This was done in order to smoothen the readings, as many sensors occasionally return incorrect readings (e.g. for a motion sensor it is common to return sequences such as "1; 2868; 2852; 1; 1; 2861; 2853", where high values indicate movement, and 1 indicates no movement).

We compared the results of three possible sensors interpretations: first interpretation always takes the latest sensor reading and considers it correct; second one takes all readings with valid lifetime and chooses the most common (average) value; third one uses RCCD in order to find the expected correct sensor readings.

The results can be seen in Table 6.2. All sensors were returning faulty readings from time to time, with M1 sensor being the least reliable (35% of erroneous readings), and PR2 sensor being the most reliable with only 1% of readings being erroneous. While averaging the value of sensors over their lifetime helped to reduce the number of erroneous sensors readings by 12.49%, the usage of RCCD to get the most probable situation reduced the number of errors by 49.78%, going from 1609 total erroroneous readings, to just 808.

The most important metric, however, is not just the total number or erroneous readings, but the total number of times the situation was detected correctly. The correctly recognized situation is the one where we know the correct values of all sensors. In our case, we update our knowledge about the environment each second, and the situation is detected correctly, if we are able to detect the correct state of all six sensors during this second. Table 6.3 shows the total number of times certain sensors gave correct readings. At no point all six sensors were giving the wrong data, but there were at least two moments, when only one out of six sensors was providing the correct data (both of which RCCD was able to detect and fix), etc. The number of times when the situation was detected fully correctly by latest sensor readings was 713 (thus sensors were fully correct 39.61% of the total time). Averaging sensor reading didn't help much, with 769 number of times (thus providing fully correct readings 42.72% of the total time). RCCD, however, was able to detect the fully correct situation 1234 number of times, which accounts for 68.56% of the total time, and is a considerably higher and better value.

Table 6.3: Times seen each number of correct sensors

Correct sensors	Latest	Averaged	RCCD
0	0	0	0
1	2	1	0
2	11	1	1
3	89	54	44
4	303	262	151
5	682	713	370
6	713	769	1234

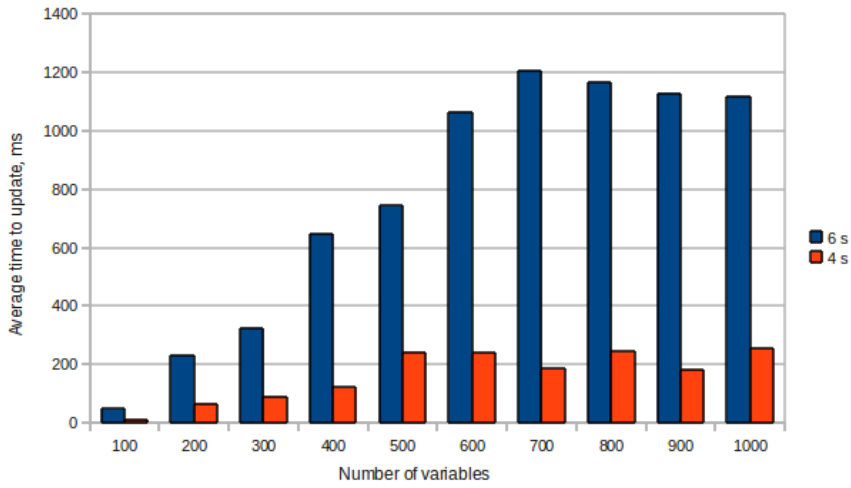


Figure 6.3: Dependence of update time on number of sensors

6.5.5 Performance

In this section, we describe how the system's performance depends on parameters of a system. The experiments were performed on a Intel Core2Duo P7370 2GHz PC with 3 GB RAM running OS Ubuntu 10.04. The software is written in Java JDK 1.6. The simulated test environment consists of a test generation part that generates situation and contexts, and a middleware part that collects contexts and maintains a diagram.

The most important among all the parameters is the size of the environment,

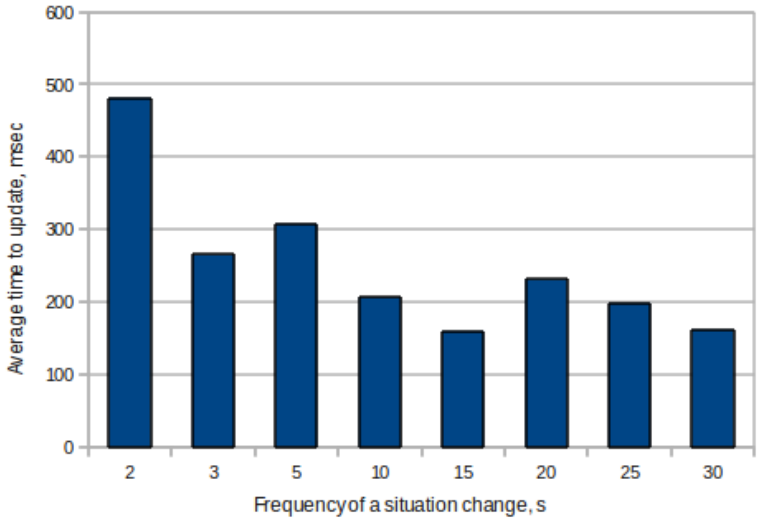


Figure 6.4: Dependence of update time on frequency of a situation change.

which is described by a number of sensors, or variables. We performed several experiments with the same parameters, while varying the number of sensors. The context arrival rate is set to 0.01 seconds; the context lifetime is set to 4 seconds in the first experiment and 6 seconds in the second one. The test generation part creates a situation. Contexts are generated based on it with a 5% error rate and are sent to the middleware part. The results can be seen in Figure 6.3. As can be seen in this figure, the average time needed to update the RCCD with each new context increases linearly with the increase of the number of sensors, until it reaches a point (in this case 700 variables) after which the time remains on the same level. This can be explained by the fact that while environments are relatively small, the increase in the environment size leads to an increase in RCCD number of nodes. However, for a certain context arrival rate and a certain context lifetime there is a certain maximum number of contexts that are usually present on a diagram simultaneously. When new contexts arrive, old contexts are removed, so the number stays on the same level. If the environment is big enough, this maximum level is reached, and beyond this the RCCD will not grow, even if environment has bigger size. Increasing the context arrival rate, or context lifetime, on the other hand, may increase this threshold. To prove it we did the same experiments, but used the context lifetime 4 seconds instead of 6 seconds, so

there was generally a smaller number of contexts on a diagram. As can be seen in Figure 6.3, the threshold was reached with smaller environment, around 500 variables, for the smaller context lifetime with the same arrival rate.

The last but not the least important parameter is the dynamicity of environment, in other words, the rate at which the situation changes. We performed several experiments with the same parameters, while changing only the frequency of situation alteration. The results can be seen in Figure 6.4. The average time of update for 2, 3 and 5 seconds is higher, while for all others, from 10 to 30 seconds, it stays approximately on the same level. The reason for this is that in all those experiments lifetime of a context was set to 6 seconds, and in the first three experiments the situation changed faster, thus leaving many obsolete contexts in a diagram and increasing its complexity. The proper solution for such cases is to decrease contexts lifetime.

Chapter 7

Policy-Based Resource Scheduling

The ability of a system to react to sudden changes in the environment, as described in Chapter 4 of this thesis, is important for any smart environment system. However, equally important is the ability to plan ahead the work of devices whenever it is possible to do so.

There are several points to think about when planning the work of devices for the near future. One of such points is the requirement to achieve the minimum necessary on-time of every device given the proper execution of device's functions. A good example is the usage of heaters only at times when they are actually needed. This achieves the minimum energy consumption while keeping users' comfort at the same level. Another point to think about is the economic impact of shifting the time of devices' activity. Questions on the economic impact are raised by advances of the research and commercialization of *smart grids*. There are many definitions of what is a *smart grid* [Morgan et al., 2009]. In a broad sense, it is an energy network where smart meters that report and control energy consumption in a fine-grained way are combined with presence of several energy providers and the ability of customers to choose the exact provider they want to get energy from.

In a system that is connected to a smart grid, the exact time of energy consumption and the exact chosen energy provider affect the total price that should be paid for consumed energy.

The system we propose utilizes an envisioned building's interface to a smart grid. In the presence of several competing energy providers and volatile prices on energy, dynamic scheduling of devices during the hours of the cheapest energy costs may considerably decrease energy bills.

The devices in the office, such as heaters, fridges, printers, projectors, are enriched with controlling modules so that their energy consumption can be measured and collected for further analysis, and devices can be controlled by an automated smart system. The system has the information about all possible device states and transitions between them that can be represented as a state machine, and the corresponding energy consumption of devices in these states. For example, a typical fridge consumes about 10^{-3} kWh when idle, but about 0.63 kWh when actively

cooling. At any moment the system has full access to the current state of a device and can trigger a state transition.

For each device, there is an associated *policy*. A policy is a set of consistent rules that must hold for proper device operations. For example, "a fridge must work at least 15 minutes per hour" to be able to maintain its internal temperature below a certain threshold temperature level.

The Scheduler receives the information from the smart grid energy providers about the available supply and price of energy. Also, the Scheduler receives the information about controllable devices, their levels of energy consumption, and their policies (rules of operation). Given this information, the Scheduler then finds the optimal solution with the minimum price paid for the total energy consumed over a certain period of time, such that all policies are adhered to.

Prices on the market change regularly, say each hour, so the Scheduler takes into account varying prices over the course of the day and tries to schedule devices to operate at times when the price per consumed kWh is the lowest. Generally, those prices vary from provider to provider, and the system can choose a provider to buy energy from. However, since providers have a finite energy supply, if many devices are scheduled to operate at the same time, their total energy consumption will likely be bigger than the cheapest energy supplier is ready to provide. That will lead to the necessity to buy energy from a more expensive energy provider.

To summarize, the Scheduler needs to balance the varying prices over the course of the day and not to schedule too many devices at the same time; thus it can avoid purchasing the more expensive energy, but at the same time keep all device policies satisfied.

7.1 Scheduling optimization problem

Let $EP(t) = \{ep_i\}$ denote a set of energy providers at the time unit t, where each *energy provider* is represented by a tuple $ep_i = \langle cost, energy \rangle$, *cost* is the cost of 1 kWh of energy, and *energy* is the maximum amount of energy that the current provider can provide at the time unit t.

To calculate the accumulated cost that the intelligent building needs to pay for the energy it consumes in a certain time unit, we sort energy providers by their price. Since we assume that a Smart Meter can choose, which provider to buy energy from, it first buys energy from the cheapest providers, and then continues to more expensive providers, if the amount of energy the building needs to consume is bigger than the amount offered by the cheapest energy providers. Thus the total cost that the building pays at time unit t if it needs to consume an amount of

Table 7.1: Example of energy providers and prices

Provider	**Energy supply**	**Price per kWh**
Internal Wind Turbine	0.214292	0.0
Internal Solar Panel	0.302314	0.122916
COMED	2.755946	0.282973
ATSI	3.154828	0.357123
AEP	2.411659	0.360658
more providers ...		

energy e is

$$cost(t, e) = min(\sum_{i=1}^{|EP(t)|} (k_i * ep_i.energy * ep_i.cost))$$

s.t.

$$\sum_{i=1}^{|EP(t)|} (k_i * ep_i.energy) = e$$

where k_i is the coefficient that shows a fraction of energy bought from energy provider ep_i. In practice, k_i will be equal to 1 for the cheapest providers, then be in a region $[0, 1]$ for one of the other providers, and be equal to 0 for all more expensive providers.

Table 7.1 shows examples of energy distribution and costs from several energy providers. Due to the absence of an actual connection to a smart grid, this data is simulated by an internal smart meter. To make a realistic simulation, the data that is used is obtained from real energy providers and prices that are provided by PJM Interconnection[1], a regional US organization that coordinates the transmission of energy in more than 13 states. The prices are obtained via a web service and are real prices for each hour of the day for the next day negotiated by energy companies.

An example of costs for the energy providers of Table 7.1 is shown in Figure 7.1. The total price to be paid equals to the area under the graph. For the consumption level of 2.1 kWh, the intelligent building can use energy from owned Wind Turbine and Solar Panels, and also buy some energy from the cheapest provider COMED, resulting in a total of $0.485217 per hour.

The algorithm to compute the cost (Algorithm 8) goes as follows. Let D denote a set of devices in the building that are connected to a Smart Meter. Each *device*

[1] `http://www.pjm.com/`

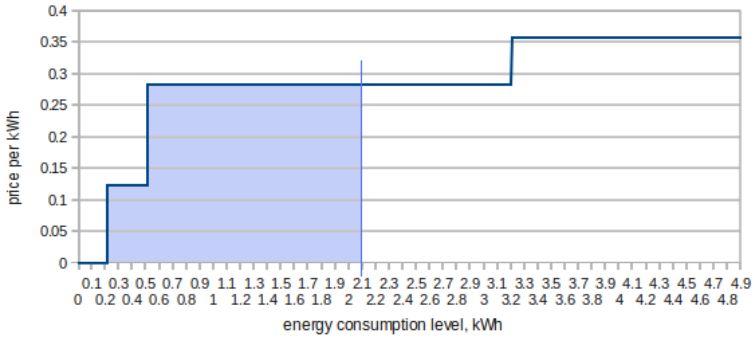


Figure 7.1: Price per kWh given the energy consumption

Algorithm 8 Cost depending on energy consumed

1: **function** getCost($time, energy$):$Double$
2: $provs$ ←getProvidersAt ($time$)
3: $sortedprovs$ ← Sort $provs$ by $provs(i).cost$
4: $energyleft$ ← $energy$
5: $totalcost$ ← 0
6: **while** $energyleft > 0$ **do**
7: $prov$ ← $sortedprovs.next$
8: $totalcost$ ← $totalcost + min(energyleft, prov.energy) * prov.cost$
9: $energyleft$ ← $energyleft - prov.energy$
10: **end while**
11: **return** $totalcost$

$d_i \in D$ is represented by a tuple $d_i = \langle did, S_i \rangle$, where did is the unique identifier of a device that in our case is equal to the device's MAC address, and S_i is a set of states that the device d_i can take (for example, "on" and "off"), where each state $s_{ij} \in S_i$ is a tuple $s_{ij} = \langle sid, energy \rangle$, sid being the unique identifier of a state, and $energy$ being an amount of energy that the device consumes while being in this state. d_{it} denotes the state that the device d_i takes at time unit $t \in T$

Let P denote a set of policies that apply to the devices in the building. Each *policy* $p_i \in P$ is a tuple $p_i = \langle did, type, params \rangle$, where did is the unique identifier of the device the policy is applied to, $type$ is the type of policy, and $params$ is a set of parameters. Parameters differ per type of policy. Each policy has different conditions that must be fulfilled in order for it to be satisfied. Specific policies that are used in this work are specified and discussed in details in Section 7.2. Here we

define a general boolean function $isSatisfied(p, X)$ that takes $true$, if the policy p is satisfied in the schedule X, and $false$ otherwise.

Over time period T, where $t \in T$ is a *time unit* in the time period T, the *schedule* $X = \{x_{td}\}$ is a set of values, where each value $x_{td} \in S_d$ represents the state that the device d takes at the time unit t. Now we can present the scheduling optimization problem:

Schedule $X = \{x_{td}\}$, $\forall t \in T, \forall d \in D$ is optimal iff

$$\sum_{t \in T} cost(t, e_t) \to min, \quad \forall p \in P : isSatisfied(p, X)$$

where $e_t = \sum_{d \in D} x_{td}.energy$.

Thus the optimal schedule is the one where price paid for all consumed energy is minimal, and the constraints for the schedule are the policies of device operation, which must be fulfilled for all devices during the scheduling period.

7.2 Policies definition

A policy is a constraint over device operation times, which must hold in order for the device to operate correctly. Examples include a fridge that must operate with certain periodicity in order to stay properly cold, or a laptop that must be plugged in for at least a certain amount of time to become fully charged.

Policies can have different parameters, a few of which are common to all, specifically $(tBegin, tEnd)$ – time period, when the policy is active; and sid – state ID that the policy is applied to. State IDs are unique per device. In general, we assume several possible states per device, together with associated actions to move a device to these states. In the presented setting, each device has two states: "on" and "off," and two associated actions: "turn on" and "turn off."

In this work, we define and use seven types of policies which represent common rules for widely deployed devices. These policies are summarized in Table 7.2 and are defined next.

TOTAL. Specifies the total amount of time tOn that a device should be put in the state sid. An example is a laptop that needs recharging for two hours. The exact time when it is going to happen does not matter, as long as it stays within $(tBegin, tEnd)$ bounds. This policy also assumes that the time when a device is in the state sid can be split into several parts. For example, we can charge a laptop for half an hour, then for another hour a little later, and for another half an hour

Table 7.2: Device policies

Policy type	Associated device	Description
TOTAL	Laptop	Device should operate for at least a certain amount of time.
CONTINUOUS		Device should operate for at least a certain amount of time uninterrupted.
REPEAT	Fridge, Boiler	Device should be put to a specified state repeatedly with a certain periodicity.
MULTIPLE	Printer	Device should operate for the time that allows for all scheduled jobs to be performed.
STRICT	Projector	A strict schedule is given in advance.
PATTERN	Microwave	An expected external pattern of device operations.
SLEEP	Any device	No demand for device during the scheduling period.

even later.

$$\forall i,\ p_i.type = TOTAL : \sum_{t=tBegin}^{tEnd} (d_{it} = sid) = tOn$$

CONTINUOUS. Similarly to the *total* policy, this one specifies the total amount of time *tOn* that a device should be put in the state *sid*. However, the *continuous* policy is stricter, as it requires that the device may not temporarily change to another state while fulfilling the policy, i.e. it must be continuously in the state *sid* for the required amount of time.

$$\forall i,\ p_i.type = CONTINUOUS :\ \exists k :\ tBegin \leq k \leq tEnd - tOn + 1$$
$$s.t.\ \forall t = tBegin..tEnd\ :\ d_{it} = sid\ \Leftrightarrow\ k \leq t < k + tOn$$

REPEAT. The device must be operated cyclically by entering the state *sid* repeatedly with a certain periodicity. For example, a fridge that should operate for 15 minutes each hour is specified using this policy. Parameters specific to this

policy are: $tCycle$ – a total cycle time; and tOn – a time during this cycle, when the device should be in a state sid.

$$\forall i,\ p_i.type = REPEAT:\ (\sum_{t=1}^{tCycle}(d_{it} = sid) = tOn)\ \bigwedge \\ (\forall j = tBegin + c..tEnd\ :\ d_{ij} = d_{i,j-c})$$

MULTIPLE. Devices that schedule a number of jobs over a certain period of time use the *multiple* policy. It has two specific parameters: $nJobs$ – a total number of jobs to be scheduled; and $tDuration$ – a time needed to complete a single job. An example is a printer that processes large batch jobs (e.g. printing a book): each job needs 15 minutes to be completed, and a total of three jobs are required to be performed. With such a policy it does not matter when a particular job is scheduled, but it is important that the device is not turned off in the middle of performing a job.

$$\forall i,\ p_i.type = MULTIPLE:\ \forall l = 1..n\ \ \exists k_l\ : \\ tBegin\ \leq\ k_l\ \leq\ tEnd - tDuration + 1\ ,\ \nexists(k_u, k_v)\ :\ |k_u - k_v| < tDuration \\ s.t.\ \forall k_l\ ,\ \forall j = 0..tDuration - 1\ \ : d_{i,k_l+j} = sid$$

STRICT. To enforce a state sid to be active from $tBegin$ to time $tEnd$, the *strict* policy is used. An example is a projector that should be turned on at the beginning of a meeting and turned off when the meeting ends. The policy firmly defines the schedule for this device, as times are strict, so the scheduler has no possibility to change the energy consumption time of the device.

$$\forall i,\ p_i.type = STRICT:\ \exists F_i(t)\ s.t.\ \forall t = tBegin..tEnd\ :\ d_{it} = F_i(t)$$

PATTERN. The *pattern* policy provides information about a way the device consumes energy. Instead of offering the possibility of controlling the device, it provides information on expected energy usage that can help to schedule other devices. For example, while a microwave is never completely turned off, the energy consumption in stand-by mode is much lower than the energy consumption when it is actively in use. Historical data that is collected during non-scheduled baseline periods shows that a higher level of energy consumption is expected during lunchtime, so the scheduler takes this into account when scheduling other devices.

This policy is similar to the *strict* one, but has some important differences. While *strict* defines exact way a device should be controlled by the system, the

pattern policy is used when device is controllable outside of the system, and only expected (statistical) information about the device operation is known. This information can be used in order to control other devices better, e.g. try to avoid turning on other devices during lunchtime, when microwave is used the most.

SLEEP. For a device for which there is no demand for operation during a given period, the *sleep* policy can be used. The policy is used mostly at night, when there is no activity in the office and many devices can be turned off in order to save energy. There are no additional parameters for this policy.

$$\forall i, p_i.type = SLEEP,\ \forall t = tBegin..tEnd:\ d_{it} = 0$$

The scheduling problem that is defined in this way is domain-independent at its core, so it can be used for other domains as long as they can be specified using similar policies as constraints to the schedule optimization. For example, in Section 7.5 we show that the core of the Scheduler can be used to schedule the deployment of different services to a cloud. The interface to the Scheduler is different for that case, and also transforms the scheduling task to the same data structure. Section 7.5 contains the details of the Schedule Interface for the clouds scenario.

7.3 Scheduler Core

The Scheduler Core is the actual implementation of the scheduling algorithm. To solve the scheduling problem, we implement a priority queue with the Breadth-First Search optimization algorithm [Russell and Norvig, 2002]. Each search state of the algorithm is a partially fulfilled schedule, and the total energy prices of partially fulfilled schedules define the search layers. We start by creating possible solutions for the first time slot and putting them to the queue. Then, at each iteration, we expand the state with the least energy cost. With each expansion, we add only those solutions that are compliant to all policies. To decrease the search space, we extensively use domain knowledge (per policy). For example, if a device has the *total* policy and should be turned on for a certain period of time, we automatically restrict from the search space all schedules where this device is turned on for more or less than the required time; this is because having it turned on more than it is absolutely necessary will only increase the energy consumption and price, and having it turned on less than absolutely necessary will not satisfy our policy. Another example is the *multiple* policy, where there are multiple jobs for a certain period of time each. We remove from the search space all schedules

Algorithm 9 Scheduler Core searching algorithm high-level overview

```
1: q ← PriorityQueue[search_node]
2: q.add(⟨0; 0; []⟩) //initialise queue with empty schedule
3: while !q.isEmpty do
4:    ⟨c; t; ps⟩ ← q.pop()
5:    R_f ← resources s.t. for the next time unit t + 1: isFeasible(R_f, t + 1, ps)
6:    for r_f ← PowerSet(R_f) do
7:       if !isAlternative(r_f) then
8:          q.add( ⟨c + cost(r_f); t + 1; ps + r_f⟩ )
9:       end if
10:   end for
11: end while
```

where time of being turned on for a device is not equal to a multiple of the time it takes to complete a single job. For example, if a single job of a printer takes 30 minutes to complete, we remove from the search space all schedules where the printer is turned on for 45 minutes, as it means the printer will definitely be idle for 15 minutes and unnecessarily consume energy.

The high-level overview of our search strategy can be seen in Algorithm 9. We create a priority queue with *a search node* that corresponds to a partially fulfilled schedule. Each search node has the following structure and is prioritized by its cost:

$$search\_node = \langle cost, time\_units, partial\_schedule \rangle$$

partial_schedule is a state matrix $partial\_schedule = T \times R$, where $T \in 1..time\_units$, and R is a set of devices. The matrix shows, for each time slot, in which state the device should be at this time slot.

The queue starts with an empty schedule. During each search step it takes the schedule with the least cost and tries to add possible distribution of resources to the next time slot. Our main contribution to scheduling strategies lies in definition of policies in such a way to drastically reduce search space. In the algorithm this is defined by two functions: *isFeasible*, which prevents from searching all schedules that breach at least one policy, and *isAlternative*, which finds if several different partial schedules actually both have the same outcome, which means that we only need to continue searching one of them, and safely drop all others.

7.3.1 Feasibility check

We decrease the search space by extensive usage of policy restrictions. For example, if a request has the *total* policy, it means it should have the available resources for a certain number of time slots, so we automatically restrict the search space to only those schedules that have this request satisfied for exactly the required number number of time slots, and remove those that have a request satisfied for more or less. Because having a request satisfied for fewer time slots means the request is not fully fulfilled. While having it satisfied for more time slots means we unnecessarily schedule more resources for usage, thus such a schedule is intrinsically not optimal. Thus, for the resource with the *total* policy we have two constraints. The first one is that the current number of time slots with scheduled resource should not exceed total expected time for resource scheduling. The second constraint is that the number of time slots left unsheduled should not exceed the difference between total expected time and current scheduled time. So, for a time slot t:

$$C_{TOTAL}: \sum_{j=tBegin}^{t} d_{ij} \leq tOn \bigwedge tEnd - t \geq tOn - \sum_{j=tBegin}^{t} d_{ij}$$

For the *continuous* policy, while searching for the optimal schedule we remove all partial schedules that assume a number of continuously used slots not equal to the total number of required time slots. All restrictions of the *total* policy are also applied to the *continuous* policy.

$$C_{CONTINUOUS}: C_{TOTAL} \bigwedge (d_{it} = 0) \Rightarrow ((\sum_{j=1}^{t} d_{ij}) = 0 \vee (\sum_{j=1}^{t} d_{ij}) = tOn)$$

For the *multiple* policy the total uninterrupted time should be divisible to the duration of one job. For example, if a job lasts two hours, and we found a partial schedule that proposed to schedule the request for three hours, we can immediately see that for one hour the request will unnecessarily occupy resources. Restrictions of the *total* policy are applicable here as well.

$$C_{MULTIPLE}: C_{TOTAL} \bigwedge (d_{it} = 0) \Rightarrow ((\sum_{j=1}^{t} d_{ij}) \; mod \; tDuration = 0)$$

The *repeat* policy is checked as the *total* one within the first cycle, and for all time slots after the first cycle, the full periodicity is applied.

$$C_{REPEAT}: \ (t < tBegin + c) \Rightarrow \sum_{j=tBegin}^{t} d_{ij} \leq tOn \bigwedge tEnd - t \geq$$

$$tOn - \sum_{j=tBegin}^{t} d_{ij}; \ (t \geq tBegin + c) \Rightarrow (d_{it} = d_{i,(t-c)})$$

The *strict* policy does not need any feasibility checking, because it is already strictly defined. There is only one way to satisfy the *strict* policy, which means it doesn't add complexity to the search space.

$$C_{STRICT}: \ d_{it} = F_i(t)$$

The *pattern* and *sleep* policies, while providing information about a device, do not require any actions to be performed on devices with these policies. Therefore such devices are removed from the search space and do not need accociated constraints.

7.3.2 Alternatives check

If the total cost of two different partial schedules is the same (which may be or not be the case, depending on energy cost for different time slots), for the purpose of finding the schedule for the next time period those two schedules are identical, if both of them contain the same number of time slots and the same number of assigned time slots for all devices. Which means we should only continue searching one of the schedules, and we can safely drop the other, as it will not produce better result.

We can only drop one of two partial schedules if (1) they have the same total cost; (2) they have the same number of scheduled time slots; (3) for each device we determine that both schedules arrive to the same current situation. The way to determine it differs per policy.

For the *total* policy, only the number of already assigned time slots matters, but not their exact position. For example, if after 30 time slots we determine that both schedules schedule a certain device for six time slots, we can regard them as the same for this request, no matter when were the exact slots when this device was scheduled. The *continuous* policy and the *multiple* policy are the same as the *total* one, we only check the total assigned time slots. The additional restrictions to the schedule are already checked at the feasibility check point, so we already know that both schedules are feasible.

We can only regard two schedules for a device with the *repeat* policy as similar in case the distribution of the assigned time within a cycle is completely the same. The reason is the distribution may matter later in the schedule, but it cannot be changed, once created during the first cycle. So two schedules for a device with the *repeat* policy are checked in the same way as for the *total* policy during the first cycle, and for all other cycles they are regarded as the same only when they really have the same assignment distribution within the first cycle.

Finally, the *strict* policy has only one possible way of being satisfied, so there is no need for additional alternatives check for this policy.

7.4 Evaluation

We have deployed the system in our own offices at the University of Groningen in order to assess the possible economic and energy savings obtainable with such a system. Our offices are located on the fifth and last floor of a more than 10000 m^2 recently erected building.[2] The test site consists of three offices occupied by permanent and PhD staff, a coffee corner/social area, and a printer area. The layout is illustrated together with the ZigBee network and the electrical appliances in Figure 7.2. In particular, we include in our testing six available devices (a fridge, a laptop, a printer, a projector, a microwave, and a water boiler). The rated power plate consumptions of the fridge and the laptop are 70 W and 90 W respectively, while that for the printer is 100 W. The projector consumes 252 W when working, while the microwave 1500 W. The water boiler consumes when heating up to 2200 W. Four other sensor nodes are also comprised in the network to strengthen the mesh network connections. We use a set of Plugwise plugs forming a wireless ZigBee mesh network around a coordinator (called "Circle+"). The network communicates with the BaseStation through a link provided by a USB stick device (called "Stick").

We have used the system over three weeks in the months of October and November 2011, and one week in the month of March 2012, performing measurements from Monday to Friday (as in the weekend there is irregular presence). In particular, in the first 2 weeks (W1-W2) we measured energy use in order to define a baseline. The third week (W3) in 2011 and the fourth week (W4) in 2012, we let the scheduling component control the environment in order to measure the actual savings. We used the *repeat* policy for the fridge (turn on for 15 minutes each hour) and the boiler (turn on for 15 minutes each two hours). The printer used the *multiple* policy, and was assigned three jobs over the course of four hours. The microwave

[2] `http://nl.wikipedia.org/wiki/Bernoulliborg`

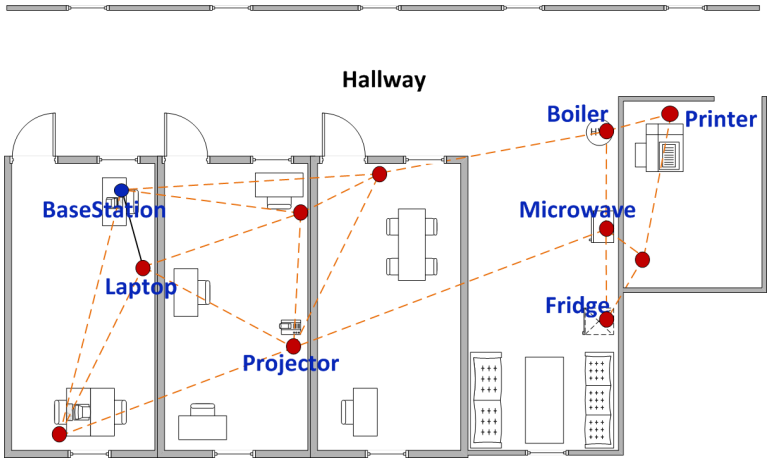


Figure 7.2: Living lab setup

used the *pattern* policy, so we used the statistical information from the previously collected data to calculate the expected level of microwave consumption at each hour of the day. The laptop used the *total* policy, so it had to be charged for a total of one hour during four hours scheduled slots. During week W3, we used the laptop each day. During week W4, we introduced variability of policies usage, so the laptop was used during Tuesday and Thursday. Projector used the *strict* policy to strictly follow the agenda of presentations. During week W3, presentations were given each day from 2 p.m. to 3 p.m. During week W4 presentations were given on Tuesday and Wednesday from 2 p.m. to 4 p.m., thus two hours each.

Next, we present the results in terms of economic savings (due to the varying prices of the smart grid) and of energy savings (due to the introduction of device policies).

7.4.1 Economic savings

The goal of the system is to save money for the office by taking advantage of the smart grid. Therefore, the first evaluation we make is based on taking the energy bill for a week using the system versus a week without it. We have considered two situations for office environments to evaluate the economic benefits of the proposed device scheduling policy: (1) an intelligent office building that interacts with the smart grid demand-response tariff service and has small scale renewable installations in its premises that provide power (W3 simulation), and (2) a more

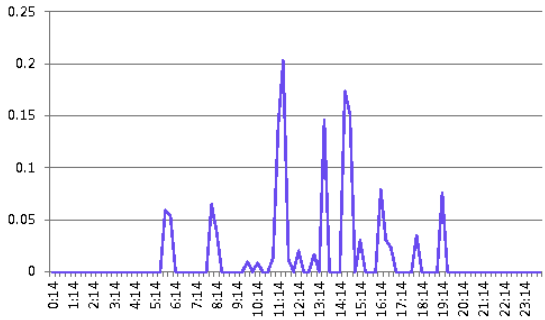


Figure 7.3: Price of energy ($ per kWh) during non-scheduled day October 27th

ordinary office that has no renewable-based power installations that provide power (W4 simulation) and that benefits only from the tariff differentiation of the smart grid. To obtain a fair comparison in the two simulations, we use the energy prices of the third week (W3) and fourth week (W4) and apply those same retrieved prices for the energy consumed in the other two weeks (W1-W2).

In the first set of simulations (office with on site small-scale renewable sources), the difference between each working day of the two weeks (average) without scheduling policies and the week where the policy has been applied is considered. The chart is shown in Figure 7.5 (top chart). It is interesting to notice the difference in the average price paid for each kWh of energy in the situation without device scheduling and, on the other hand, considering scheduling. On average, the price in $ per kWh drops by more than 27% in the two situations. An interesting day where the savings on energy expenses are particularly significant is between the three consecutive Thursdays monitored (October 20th, 27th, and November 3rd). Comparing these three days, the money savings are on average more than 50%. A comparison between the price paid for energy in each hour between the situation in October 27th and November 3rd is shown in Figures 7.3 and 7.4, respectively. In particular, one can see the cut-off of unnecessary energy expenses related to those consumptions that happen during non-working time (late evening or during the night) by devices that are not strictly necessary (most notably the hot water boiler). Another optimization the system achieves is the most efficient schedule of devices, when the energy generated by photovoltaic panel is more intense and whose cost is generally smaller than energy provisioning on the market.

To validate the scheduling policy, in W4 we consider an office without renewable energy sources (whose price is generally cheaper than energy provision market). Results comparing the day-by-day average price between the scheduling situation

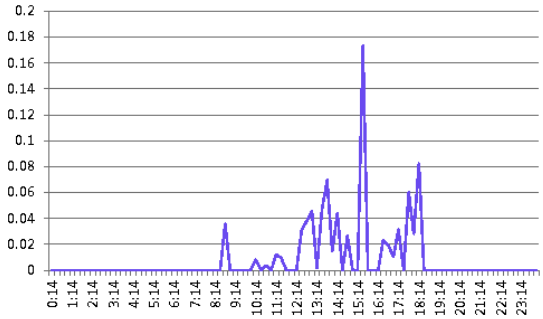


Figure 7.4: Price of energy ($ per kWh) during scheduled day November 3rd

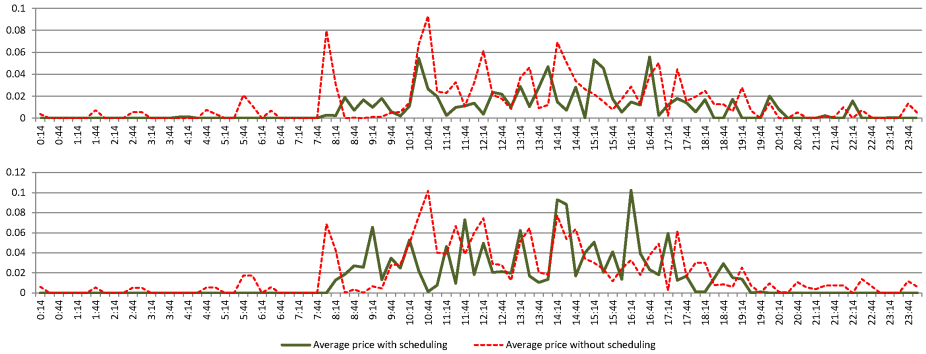


Figure 7.5: Average price ($ per kWh) comparison between scheduled (continuous line) and non-scheduled (dashed line) situations (top is W3, bottom is W4)

and the non-scheduling one are shown in Figure 7.6, while the daily average is shown in Figure 7.5 (bottom chart). One can see that the average price paid when scheduling is active is usually lower than the non-scheduled situation (cf. the continuous and dashed line in Figure 7.5); the overall economic savings between the situation when the schedule is implemented and when it is not is about 22%. The lower savings compared to the W3 experiment are due to the absence of renewable sources in the energy mix of the office, which we have assumed cheaper than the traditional energy market provider prices.

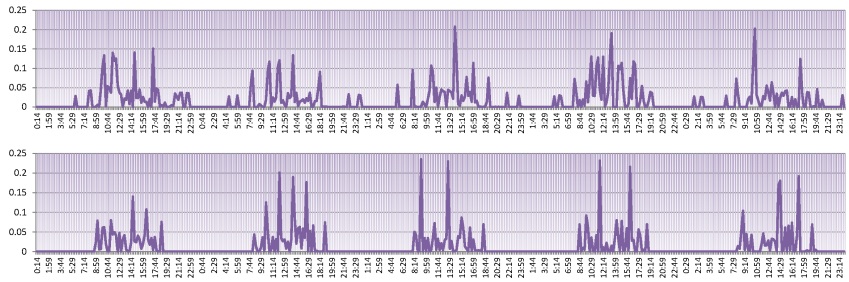


Figure 7.6: Average price ($ per kWh) comparison between non-scheduled (upper chart) and scheduled (lower chart) appliances for each work day (W4 experiment)

7.4.2 Energy Savings

Although energy use reduction is not the primary aim of the system, but rather economic savings based on dynamic pricing, the use of policies for devices alone provides for energy saving in absolute terms. The scheduling reduces the consumption of devices that are not used during non-working hours and that do not impact the habits of the users (e.g. keeping the hot water boiler working at night); in addition, the Scheduler tries to use at best the cheap electricity coming from the solar panels during daylight hours. Figure 7.7 visually reinforces the idea of reducing loads when unnecessary among the normal (first upper chart) and the scheduled solutions (the middle and bottom charts): one notices a more compact chart in which energy is used mostly during daytime (8 a.m.-6.30 p.m.) in each day of the week. The average savings of energy consumed between the situation without the scheduling policy and the situation considering it, is more than 15% (W1-2 versus W3 experiment) and about 11% (W1-2 versus W4 experiment), respectively. We ascribe the small difference in percentage to the unpredictable usage of equipment in the actual living lab between the two weeks (e.g. microwave use).

7.4.3 Discussion on System Performance

Finding the optimal schedule for a set of devices is a computationally expensive problem and while there exist many tools that can solve such problems reasonably fast for practical domain sizes, we took a set of measures to ensure that our solution will remain within practical bounds for bigger lab settings. There are three dimensions that determine the input size of the scheduling task: number of energy providers, time period of the schedule, and number of devices.

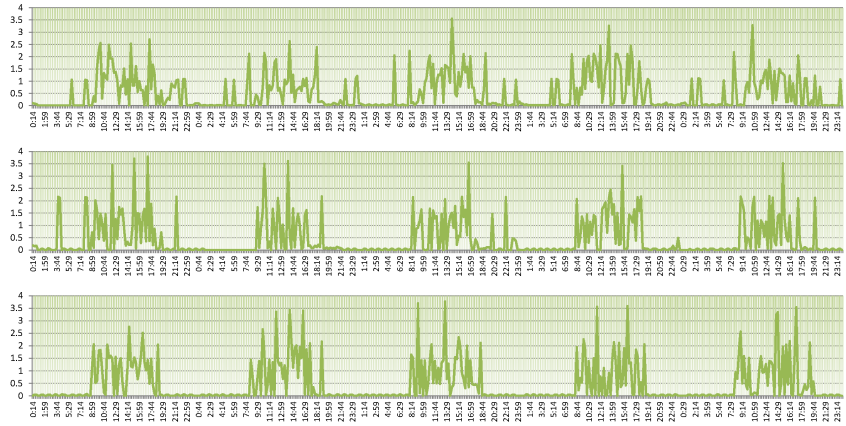


Figure 7.7: Energy (per kWh) comparison between non-scheduled (upper chart) and scheduled appliances (middle and bottom chart respectively W3 and W4)

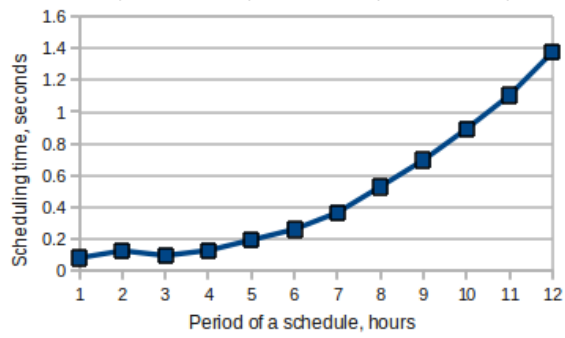


Figure 7.8: Scheduler performance dependence on the schedule time period

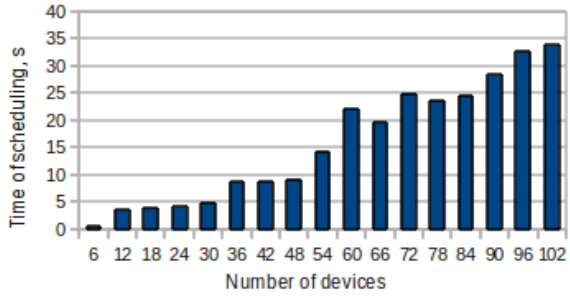


Figure 7.9: Scheduler performance dependence on the number of devices

The increase in the *number of energy providers* has negligible impact on the performance of the Scheduler. The reason for this is that the function of price levels has to be computed only once at the beginning of the scheduling task, as described in Algorithm 8. During the actual schedule search, we refer to the pre-calculated function, and the time for such a referring does not depend on the original number of energy providers.

The time it takes for the Scheduler to find the optimal schedule grows with the *number of time units*, for which we are obtaining a schedule. We tried to vary the time period of the schedule from 1 hour to 12 hours; the average length of the Scheduler running time is shown in Figure 7.8. As can be seen in this figure, even for 12 hours period it takes only about 1.4 seconds to find the optimal schedule for our living lab setting.

The *number of devices* causes the biggest strain on the system's performance. Since in the living lab we had only 6 devices, to test the Scheduler with a larger number of them, we simulated devices by creating multiple copies of the available devices. We determined that a search for the optimal schedule can take an impractically large amount of time for large centrally controlled buildings. This is less of a problem that it might initially appear to be. In fact, one can dynamically relax the requirement for optimality and search instead for a "good enough" schedule. For our scheduling algorithm, we implemented a gradual approach. For a large number of devices, we divide them into groups of approximately equal size. We run the Scheduler for the first group, and find the optimal solution for it. Then, given this schedule for the first group (which we do not change while scheduling the other groups), we calculate the increased amount of energy used at each time unit, and we run the Scheduler for the next group of devices, finding the optimal solution for them. After this, we recalculate the increased amount of energy again and run the Scheduler for the third group, and so on, until all devices are scheduled. Note that, while this approach follows a greedy practice, the schedule provided is still quite efficient in terms of price savings and smart distribution of devices working time. If the devices from the first group were scheduled to run at a certain time unit, the amount of energy already consumed at this time will be large; this will prevent the Scheduler from placing more devices from the second group in the same time slot. So the Scheduler is still able to distribute the working time of devices across different time units even for devices from different groups. In Figure 7.9, we show the averaged running time of the Scheduler for a different number of devices.

7.5 Digression: Scheduler Service Interface for Clouds

The core of the Scheduler that is described in Section 7.3 is designed to be domain-independent. Due to this we were able to utilise the Scheduler in a different setting, namely for optimizing deployment of services to the cloud, i.e. a distributed computing network.

The Scheduler is responsible for optimization of the total number of resources required to satisfy requests of different services to be deployed to the cloud with particular resources available to them. The Scheduler service has a REST interface and is invoked at the beginning of the time interval to be scheduled (usually a week), with all requests for the following time interval. The requests are stored and passed in Google Protocol Buffers[3] format.

The Scheduler service has two parts: Cloud Schedule Interface (CSI) and Scheduling Core. Cloud Schedule Interface is a wrapper on top of the Scheduler Core and is specific to the cloud scheduling optimization problem. The task of this component is to transform cloud schedule requests to the domain-independent representation within Scheduling Core, and to transform back the resulting schedule to the required form. The Scheduling Core itself is domain-independent, and can be used within any domain, as long as constraints of scheduling are specified in similar policy types. Therefore, the same software module that was used for the smart environment devices was reused for this task.

As input to the optimization task, the Scheduler receives a list of *requests*.

Definition 9 (Request)**.** A *request* is a full specification of the number and type of *resources* needed to satisfy a certain task, together with the *policy* of resources usage.

The informal examples of requests may be "five servers are needed for a total of eight hours running time to execute Services X, Y, and Z", or "three servers are needed to run continuously for twelve hours to execute Service K as a shared service." To satisfy the request, the requested number of resources should be available for the required amount of time. Partial satisfaction of a request is not possible (since partially satisfied request means an incomplete task). A request should either be satisfied fully, or not at all.

Thus, to fully define the request, we first need to define its two most important parts: the list of *resource demands* and the execution *policy*.

[3] `code.google.com/apis/protocolbuffers`

Definition 10 (Resource demand)**.** A *resource demand* is the specification of a resource that is needed to complete the task, which includes the specification of the type of service which should be running, the number of services, and whether the service can be shared with other tasks, or must be run exclusively.

The data model uses Google Protocol Buffers format, where all variables in a message are described as a tuple: *"modifier, type, variable name = parameter id"*. In the code presented here, we omit *parameter id* for clarity purposes.

The data for resource demand written as follows:

```
message ResourceDemand
  optional uint32 resId;
  optional uint32 number;
  optional bool   shared;
```

Here the "*resId*" uniquely represents the service to run, "*number*" is the number of such services that should be started, and a boolean value "*shared*" represents whether the service can also be used by other requests, or should be run exclusively by this request.

While by using a list of resource demands we can specify all the resources that we need, we also should specify the time frame for them to run, and the execution policy. For example, one task may require for services to be run for twelve consecutive hours, while another task may require them to run for twenty four hours, while not caring whether those hours are consecutive or split apart.

Therefore we use the same policies as for the Smart Home scenario, as described below. For all policies, we define T as the total number of time slots, p_{ij} as the scheduled status of request r_i at time slot j ($p_{ij} = true$, if request r_i is scheduled for the time slot j, and $p_{ij} = false$ otherwise).

Total. The policy has an additional parameter d ("*duration*"), and assumes that resources should be available for the total number of time slots, equal to the "*duration*" value. How the time slots are split over the whole scheduling period is not important, thus the task can be split and, for example, it can run on Monday, Wednesday, Friday, or on Monday to Wednesday.

Continuous. The policy is stricter, and guarantees that once the request is started, it will run *uninterrupted* for the required number of time slots, also specified by the parameter d ("*duration*").

Multiple. The policy allows for more that one job to be scheduled within the same request. Each job must have resources within uninterrupted period of time, but jobs themselves may be split in time, for example, one job can be executed on

Monday, and two more on Thursday. In addition to the d ("*duration*") of the job parameter, the policy also has a n ("*number of jobs*") parameter.

Repeat. The policy has two parameters: c ("*cycle duration*") and d ("*total time to be scheduled within a cycle*"), and assumes that a resource should be available cyclically with a certain periodicity. Example are regression tests that must be run for an hour every day (to test nightly builds).

Strict. The policy firmly defines the specific schedule for certain resource requests. Thus these resource requests cannot be moved to different time slots, but the knowledge about them allows Scheduler to schedule other requests to share resources with the strictly defined requests, whenever possible.

Note that we do not use the *pattern* and the *sleep* policies for the services to cloud deployment scenario. The *pattern* policy represents statistical expected behavior of a device. A service request, on the other hand, can always be scheduled exactly the way the system wants, and therefore the *strict* policy is used for these purposes. The *sleep* policy represents absense of any specific deployment needs, i.e. a service does not require to be scheduled for deployment at all. As such, a particular request for this service can be fully removed from the scheduling (unlike a device, which cannot be easily removed from the system), i.e. there is no need for the *sleep* policy for service to cloud deployment setting.

Thus, the data model to specify the policy is the following:

```
enum PolicyType
  TOTAL;
  CONTINUOUS;
  MULTIPLE;
  REPEAT;
  STRICT;

message Policy
  required PolicyType type;
  optional uint32     duration;
  optional uint32     numberJobs;
  optional uint32     cycleDuration;
  repeated uint32     strictTimeOn;
```

Now that we have specified both the policy data model and the resource demands data model, we can fully specify the request:

```
message Request
  required uint32 reqId;
  optional Policy policy;
  repeated ResourceDemand demand;
```

Note that each request can contain a list (specified by the keyword "*repeated*") of different resource demands, and to satisfy the request, all resource demands must be satisfied at the same moment in time.

To create a schedule of requests, some additional information is also needed in addition to a list of requests. First of all, the number of available time slots over the whole scheduling period should be given. Furthermore, the total number of resources available at the same time should be specified. If resources represent a single server instance in a cloud, it is usual for the cloud providers to charge more per server, if many servers are used at the same time. Additional costs can be avoided in case we limit our execution by not using more than a certain number of machines at each moment in time. For a different scenario, if we execute services not in a cloud, but on actual physical servers that are available to us, the number of such servers is also limited, which should be taken into account by the Scheduler. Thus, the schedule request data model is the following:

```
message ScheduleRequest
  repeated Request reqList;
  required uint32  numSlots;
  required uint32  numResources;
```

"*reqList*" is the list of requests, "*numSlots*" is the number of available time slots (usually an hour, but can also be any other time interval), "*numResources*" is the maximum number of resources that can be used at the same time.

As a result of the Scheduler execution we obtain a full schedule of requests distributed over available time slots. For each time slot, the Scheduler presents a list of request IDs to show which requests should run at this time. The data model for the Scheduler response is the following:

```
message ScheduledTimeSlot
  repeated uint32 reqList

message Schedule
  repeated ScheduledTimeSlot schedule;
```

We can optimize the resource usage by maximizing the reuse of shared resources. If requests require same shared resources, placing them at the same time slots will enable maximum reuse.

Chapter 8

Conclusions

In this dissertation we investigated the usage of constraints and logical rules for reasoning and decision making in smart environments. We presented an implementation of the corresponding system, and showed its results in several living lab experiments.

We started by overlooking past and present smart environment projects, and their architectures in particular. We documented the observed architecture patterns in Chapter 3. These patterns helped to define a place of a reasoning module within a big picture of the whole system.

In Chapter 4 we presented a reactive reasoning module that is based on dynamic constraint satisfaction principles. Because the module can dynamically keep track of parts of the environment and behavior rules that are affected by sensor changes, the module can only recheck those affected parts, while still keeping the global environment satisfied and globally optimal. This approach is proved to consistently outperform straightforward CSP approach, which allows to scale the reasoning system to much bigger sizes of environments.

Most reasoning systems are vulnerable to missing or faulty sensor data. Therefore in Chapters 5 and 6 we investigated the data structure which can help us to use interdependency rules between sensors to resolve contradictions in data, and to reduce ambiguity if the data is incomplete.

And finally, to augment the immediate reactions to the changes in the environment with a behavior that produces smarter solutions over time, we also investigated the scheduling of devices in a building connected to a smart grid.

Based on this dissertation we can now answer the research questions that were raised in Section 1.1.

RQ1. *What are the commonalities in the design and development process of smart environments? Can any pattern be derived from technical architectures of such systems? How can the process be streamlined, made easier? Which knowledge from existing projects can be reused in new projects?*

As was shown in Chapter 3, many independently constructed smart environ-

ments nevertheless share a large amount of common features in technical architectures of their projects. Layered approach to the architecture structure is arguably the most natural way to arrange different modules by their responsibilities and interconnections. We have described four main layers that are inevitably present in some form in most of smart environment projects. The composition of these components can be presented as a commonly used technical architecture pattern for such projects.

By the nature of smart environments, physical sensors are an inherent part of them. Therefore the Physical layer is a mandatory part of the architecture and contains all hardware devices and low-level software protocols to deal with these devices. Often a multi-faceted smart environment requires diverse sensors and actuators to function properly. These diverse devices all operate via their own protocols, therefore a common gateway that hides the complexity from all other parts of the system and unifies the collection of information, while strictly speaking being optional, is also a very common sight as a part of the Physical layer.

The Ubiquitous layer is the backbone of the system and oversees main information flows, acts as a storage for knowledge base, and supports other components by providing access to the parts of the system they need. It is customary for higher-level layers of smart systems to use a high-level domain information and representation for reasoning. The components that are responsible for collecting raw sensor data and translating it to the domain-level representation are a part of the Ubiquitous layer. Common components for this layer include Knowledge Base and Context component for processing sensor data. For systems that have actuators, the Execution component that issues the low-level commands to the actuators is also a part of this layer.

The Reasoning layer is a very diverse layer among smart environment projects. The components from this layer are what makes a smart environment smart. They come in many different flavors, and include components for learning, activity recognition, decision making, planning and scheduling, etc. The exact composition of components is dictated by the requirements of the project, however the common information flow patterns include communication with the Ubiquitous layer for data collection and external events processing and communication with the User layer for reporting and system's goals updates.

The User layer contains all components that are responsible for domain-level communication with users, presenting them with information about system's status and decisions, and allowing them to change the goals and behavior of the system according to their requirements.

The architecture pattern that is described in Chapter 3 can be used by new

smart environment projects as a proven foundation to build upon and add features that are unique to these project. By reusing this knowledge, the design and development process of new smart environment projects may be streamlined and made easier.

RQ2. *What is an effective approach to design a reasoning engine for smart environments that fulfills all important requirements (e.g. scalability, robustness, dynamic adaptation, computational efficiency, real-time response, and so on)? Is there any specific structure or some distinguished features of smart environment domains? If yes, can this specific structure be exploited to increase the performance and/or reasoning capabilities of a reasoning engine operating with such domains?*

This research question investigates the problem of finding the best environment state that conforms to all constraints that appear due to physical environment configuration and due to different preferences of users. Chapter 4 showed that such a problem can be easily modelled as a constraint satisfaction problem (CSP). However, the straightforward CSP task does not fulfill the requirements for scalability and computational efficiency, because of multiple unnecessary computations that need to be performed after every new sensor change. Since in a dynamic environment there may be many sensor changes per second, the system may not be able to respond to changes in real-time.

Therefore Chapter 4 introduced the dependency graph data structure that dynamically keeps track of interdependent parts of the environment. After every sensor change, the dependency graph provides information on which parts of the environment and which rules are affected by it. It is formally proven in the chapter that it is possible to only recheck the affected part while still keeping the full environment globally satisfied and optimal.

The experiments performed in a real living lab environment and in simulations proved that the usage of dependency graph consistently outperforms the straightforward CSP task. The solution for real environments is able to compute the optimal answer in real-time, therefore it complies to the requirements of scalability and computational efficiency.

The description of real smart environments shows that such environments contain *clusters* of variables, with high level of dependency among variables within a cluster, and loose dependency of those variables on variables outside the cluster. Clusters usually contain variables within a single physical space, such as a room or a single working desk, and sometimes clusters can be split by nature of variables, for example variables that affect lighting system can form a cluster. The experiments performed with the solution based on the dependency graph showed that the performance increases with more distinctly defined clusters.

RQ3. *How can the effect of sensor errors be minimized with respect to decision making? Can a reasoning engine work with incomplete and/or conflicting sensor data? If there is no definite answer on which data is incorrect, can the system operate correctly in presence of conflicting data?*

Chapter 5 introduced a context consistency diagram, a data structure that allows to reason about the current state of the environment even in a presence of sensor errors, incomplete or conflicting sensor data. By explicitly capturing the dependencies between different sensors it is possible to understand if several sensor readings support the same view on the current situation, or if some sensor readings contradict each other. The chapter showed that in order to minimize the effect of sensor errors, it is important to keep all incoming data even if there are conflicts or inconsistencies in it. If the inconsistency is resolved incorrectly, the correct data may be discarded instead of the incorrect one, worsening the initial situation and decision making capabilities. The context consistency diagram allows to store all incoming data, and to see, which sensor readings support each other, and which are inconsistent with each other. By assigning weights to initial sensor readings, and rewarding them for being consistent, it is possible to calculate probabilities for all possible situations of being correct at the current moment. Since no data is lost, if further sensor readings support the situation that initially had a lower probability, this situation may become the most probable one with time and new information.

The context consistency diagrams allow different type of queries, and can provide the probability that a particular situation is true, the probability that a variable has a certain value, or the conditional probability of a certain situation or a certain value of a variable if another variable has a priori known value. Chapter 6 described a reduced context consistency diagram, which is much smaller than the full one, but has lower querying capabilities and can only answer to a question which situation is the most probable at this moment in time.

The cautious approach of the context consistency diagrams that keeps the data about all possible interpretations, allows also to include precautionary actions about critical situations even if they are not the most probable ones, for example, raise an alarm if the probability of fire is more than 20%.

Several experiments performed with real sensor data showed that the CCD can solve around 40% of initial sensor reading errors, effectively improving the recognition rate of the actual situation.

RQ4. *How can a smart system utilize the existence of diverse energy providers in order to minimize the cost of energy over time? Does this smart system affect*

total energy consuption? Which information should be available to a reasoning engine in such a case, and how to use it in the optimal way?

As shown in Chapter 7, smart buildings contain a lot of devices with loose dependency on immediate outside conditions, which may be activated at the smart system's discretion at different times as long as their activity conforms to certain constraints, or device policies, such as the fridge must be turned on and actively cooling at regular intervals, or the laptop must be plugged in and charging for a certain total amount of time in order to be fully charged.

A building that is connected to a smart grid can obtain energy from diverse energy providers, which may change the amount of available energy and its price at different times. Chapter 7 describes the scheduling mechanism that can utilize information from different providers in order to create a schedule for controllable devices of the system such that this schedule conforms to all restrictions of particular devices and their policies of operation are satisfied, and the price paid for the energy consuption of those devices is minimal.

The Scheduler is designed as an optimization task with additional constraints given by the policies of devices. Every policy has associated feasibility check and a check for alternatives, where the feasibility check answers if the partial schedule conforms to the policy of the device, and the check for alternatives keeps only one partial schedule out of several that all provide the same total results. These checks allow to restrict the search space of the optimization algorithm, making it more scalable.

The experiments on the real environments showed that scheduling allows to reduce the price paid for the energy by up to 50%. Due to scheduling all devices for the minimum uptime that still keeps their policies satisfied, the scheduling also allows to save up to 10% of energy consumption comparing to an environment without automated scheduling.

Most of the topics covered by this thesis are open to further research. For example, the system that is described in Chapter 4 of the thesis does not cover the automated learning of rules, even though such learning may greatly reduce efforts needed for initial system's deployment. In this respect, our further works [Degeler et al., 2014] that do investigate this topic can be seen as further extension of the research done in this dissertation.

Bibliography

Abowd, G. D., Dey, A. K., Brown, P. J., Davies, N., Smith, M. and Steggles, P., (1999). Towards a better understanding of context and context-awareness, *in Handheld and ubiquitous computing*, Springer, pp. 304–307.

Aiello, M., Aloise, F., Baldoni, R., Cincotti, F., Guger, C., Lazovik, A., Mecella, M., Pucci, P., Rinsma, J., Santucci, G. et al., (2011). Smart homes to improve the quality of life for all, *in Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*, IEEE, pp. 1777–1780.

Aloise, F., Schettini, F., Aricò, P., Salinari, S., Guger, C., Rinsma, J., Aiello, M., Mattia, D. and Cincotti, F., (2011). Asynchronous p300-based brain-computer interface to control a virtual environment: Initial tests on end users, *Clinical EEG and Neuroscience* 42(4), 219–224.

Amft, O. and Lombriser, C., (2011). Modelling of distributed activity recognition in the home environment, *in Int. Conf. Engineering in Medicine and Biology Society (EMBC)*, IEEE, pp. 1781–1784.

Antoniou, G. and van Harmelen, F., (2009). Web ontology language: Owl, *in Handbook on ontologies*, Springer, pp. 91–110.

Baldauf, M., Dustdar, S. and Rosenberg, F., (2007). A survey on context-aware systems, *International Journal of Ad Hoc and Ubiquitous Computing* 2(4), 263–277.

Bessiere, C., (1991). Arc-consistency in dynamic constraint satisfaction problems., *in AAAI*, Vol. 91, pp. 221–226.

Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A. and Riboni, D., (2010). A survey of context modelling and reasoning techniques, *Pervasive and Mobile Computing* 6(2), 161–180.

Bliek, F., van den Noort, A., Roossien, B., Kamphuis, R., de Wit, J., van Der Velde, J. and Eijgelaar, M., (2010). Powermatching city, a living lab smart grid demonstration, *in Innovative Smart Grid Technologies Conference Europe (ISGT Europe), 2010 IEEE PES*, IEEE, pp. 1–8.

Brown, P. J., Bovey, J. D. and Chen, X., (1997). Context-aware applications: from the laboratory to the marketplace, *Personal Communications, IEEE* 4(5), 58–64.

Bu, Y., Chen, S., Li, J., Tao, X. and Lu, J., (2006). Context consistency management using ontology based model, 4254, 741–755.

Bu, Y., Gu, T., Tao, X., Li, J., Chen, S. and Lu, J., (2006). Managing quality of context in pervasive computing, *in QSIC '06: Proceedings of the Sixth International Conference on Quality Software*, IEEE Computer Society, pp. 193–200.

Callaghan, V., Clarke, G., Colley, M., Hagras, H., Chin, J. and Doctor, F., (2004). Inhabited intelligent environments, *BT Technology Journal* 22(3), 233–247.

Capodieci, N., Pagani, G. A., Cabri, G. and Aiello, M., (2011). Smart meter aware domestic energy trading agents, *in Proceedings of the 2011 workshop on E-energy market challenge*, ACM, pp. 1–10.

Castelli, G., Rosi, A., Mamei, M. and Zambonelli, F., (2006). The w4 model and infrastructure for context-aware browsing the world., *in WOA*.

Cesta, A., Cortellessa, G., Oddi, A., Policella, N. and Susi, A., (2001). A constraint-based architecture for flexible support to activity scheduling, *in AI* IA 2001: Advances in Artificial Intelligence*, Springer, pp. 369–381.

Chen, G., Kotz, D. et al., (2000). A survey of context-aware mobile computing research, Technical report, Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College.

Chodorow, K. and Dirolf, M., (2010). *MongoDB: the definitive guide*, O'Reilly Media.

Cook, D. and Das, S., (2007). How smart are our environments? An updated look at the state of the art, *Pervasive and Mobile Computing* 3(2), 53–73.

Cook, D. J., (2009). Multi-agent smart environments, *Journal of Ambient Intelligence and Smart Environments* 1(1), 51–55.

Das, S., Cook, D., Battacharya, A., Heierman III, E. and Lin, T., (2002). The role of prediction algorithms in the mavhome smart home architecture, *Wireless Communications, IEEE* 9(6), 77–84.

Debruyne, R., (1996). Arc-consistency in dynamic CSPs is no more prohibitive, *in IEEE Int. Conf. Tools with Artificial Intelligence (ICTAI)*, pp. 299–306.

Dechter, R. and Dechter, A., (1988). *Belief maintenance in dynamic constraint networks*, University of California, Computer Science Department.

Dechter, R. and Pearl, J., (1987). Network-based heuristics for constraint-satisfaction problems, *Artificial Intelligence* 34(1), 1–38.

Degeler, V., Gonzalez, L. I. L., Leva, M., Shrubsole, P., Bonomi, S., Amft, O. and Lazovik, A., (2013). Service-oriented architecture for smart environments, *in IEEE International Conference on Service Oriented Computing and Applications (SOCA 2013)*.

Degeler, V. and Lazovik, A., (2011). Interpretation of inconsistencies via context consistency diagrams, *in Pervasive Computing and Communications (PerCom), 2011 IEEE International Conference on*, IEEE, pp. 20–27.

Degeler, V. and Lazovik, A., (2012*a*). Cost-efficient context-aware rule maintenance, *in IEEE Context Modeling and Reasoning (CoMoRea), Pervasive Computing and Communications (PerCom) Workshops*, IEEE, pp. 608–612.

Degeler, V. and Lazovik, A., (2012*b*). Reduced context consistency diagrams for resolving inconsistent data, *ICST Transactions on Ubiquitous Environments* 12(10-12).

Degeler, V. and Lazovik, A., (2013*a*). Architecture pattern for context-aware smart environments, *Creating Personal, Social and Urban Awareness through Pervasive Computing. IGI Global* pp. 108–130.

Degeler, V. and Lazovik, A., (2013*b*). Dynamic constraint reasoning in smart environments, *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)* .

Degeler, V., Lazovik, A., Leotta, F. and Mecella, M., (2014). Itemset-based mining of constraints for enacting smart environments, *in Proceedings of the Symposium on Activity and Context Modeling and Recognition*.

Dey, A. K., Abowd, G. D. and Salber, D., (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications, *Human-computer interaction* 16(2), 97–166.

Du, P. and Lu, N., (2011). Appliance commitment for household load scheduling, *Smart Grid, IEEE Transactions on* 2(2), 411–419.

Fielding, R. T. and Taylor, R. N., (2002). Principled design of the modern web architecture, *ACM Transactions on Internet Technology (TOIT)* 2(2), 115–150.

Franklin, D. and Flaschbart, J., (1998). All gadget and no representation makes jack a dull environment, *in Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments*, pp. 155–160.

Georgievski, I., Degeler, V., Pagani, G. A., Nguyen, T. A., Lazovik, A. and Aiello, M., (2012). Optimizing energy costs for offices connected to the smart grid, *IEEE Transactions on Smart Grid* 3, 2273–2285.

Henricksen, K. and Indulska, J., (2004). Modelling and using imperfect context information, *in Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, IEEE Computer Society, p. 33.

Huang, Y., Ma, X., Cao, J., Tao, X. and Lu, J., (2009). Concurrent event detection for asynchronous consistency checking of pervasive context, pp. 1 –9.

Huang, Y., Ma, X., Tao, X., Cao, J. and Lu, J., (2008). A probabilistic approach to consistency checking for pervasive context, *in EUC '08: Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, IEEE Computer Society, pp. 387–393.

Jeffery, S. R., Garofalakis, M. and Franklin, M. J., (2006). Adaptive cleaning for RFID data streams, *in Proceedings of the International Conference on Very Large Data Bases*, Vol. 1, pp. 163–174.

Jussien, N., Rochart, G., Lorca, X. et al., (2008). Choco: an open source java constraint programming library, *in CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pp. 1–10.

Kaldeli, E., Warriach, E. U., Bresser, J., Lazovik, A. and Aiello, M., (2010). Integrating, composing and simulating services at home, *in International Conference on Service Oriented Computing (ICSOC)*.

Kaldeli, E., Warriach, E. U., Lazovik, A. and Aiello, M., (2013). Coordinating the web of services for a smart home, *ACM Transactions on the Web (TWEB)* 7(2), 10.

Koes, M., Nourbakhsh, I. and Sycara, K., (2006). Constraint optimization coordination architecture for search and rescue robotics, *in Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, IEEE, pp. 3977–3982.

Kok, K., Derzsi, Z., Hommelberg, M., Warmer, C., Kamphuis, R. and Akkermans, H., (2008). Agent-based electricity balancing with distributed energy resources, a multiperspective case study, *in Hawaii international conference on system sciences, proceedings of the 41st annual*, IEEE, pp. 173–173.

Kong, H., Xue, G., He, X. and Yao, S., (2009). A proposal to handle inconsistent ontology with fuzzy owl, *in Proc. WRI World Congress on CS and Inf. Eng.*, Vol. 1, pp. 599–603.

Kreucher, C., Blatt, D., Hero, A. and Kastella, K., (2006). Adaptive multi-modality sensor scheduling for detection and tracking of smart targets, *Digital Signal Processing* 16(5), 546–567.

Kusznir, J. and Cook, D., (2010). Designing lightweight software architectures for smart environments, *in Intelligent Environments (IE), 2010 Sixth International Conference on*, IEEE, pp. 220–224.

Lakshman, A. and Malik, P., (2009). Cassandra: A structured storage system on a p2p network, *in Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, ACM, pp. 47–47.

Lassila, O., Swick, R. R. et al., (1998). *Resource Description Framework (RDF) model and syntax specification*, Citeseer.

Loke, S., (2006). *Context-aware pervasive systems: architectures for a new breed of applications*, CRC Press.

López-de Ipiña, D., Almeida, A., Aguilera, U., Larizgoitia, I., Laiseca, X., Orduña, P., Barbier, A. and Vazquez, J., (2008). Dynamic discovery and semantic reasoning for next generation intelligent environments, *in Intelligent Environments, 2008 IET 4th International Conference on*, IET, pp. 1–10.

Lu, H., Chan, W. and Tse, T., (2008). Testing pervasive software in the presence of context inconsistency resolution services, *in ICSE '08: Proceedings of the 30th international conference on Software engineering*, ACM, pp. 61–70.

Mackworth, A. K., (1977). Consistency in networks of relations, *Artificial intelligence* 8(1), 99–118.

Marcelloni, F. and Aksit, M., (2001). Leaving inconsistency using fuzzy logic, *Information and Software Technology* 43(12), 725 – 741.

Mittal, S., Aggarwal, A. and Maskara, S., (2012). Situation recognition in sensor based environments using concept lattices, *in Proceedings of the CUBE International Information Technology Conference*, ACM, pp. 579–584.

Mittal, S. and Falkenhainer, B., (1990). Dynamic constraint satisfaction, *in Nat. Conf. on Artificial Intelligence*, pp. 25–32.

Morgan, M. G., Apt, J., Lave, L., Ilic, M. D., Sirbu, M. A. and Peha, J. M., (2009). The many meanings of "Smart Grid".

Neves-Silva, R., Ruzzelli, A., Fuhrmann, P., Bourdeau, M., Pérez, J. and Michaelis, E., (2010). Energy consumption prediction from usage data for decision support on investments: The enprove approach, *in Control Methodologies and Technology for Energy Efficiency*, Vol. 1, pp. 48–52.

Nguyen, T. A. and Aiello, M., (2012). Beyond indoor presence monitoring with simple sensors., *in PECCS*, pp. 5–14.

Nguyen, T. A. and Aiello, M., (2013). Energy intelligent buildings based on user activity: A survey, *Energy and buildings* 56, 244–257.

Nguyen, T. A., Degeler, V., Contarino, R., Lazovik, A., Bucur, D. and Aiello, M., (2013). Towards context consistency in a rule-based activity recognition architecture, *in International Symposium on Ubiquitous Intelligence and Autonomic Systems*.

Nizamic, F., Degeler, V., Groenboom, R. and Lazovik, A., (2012). Policy-based scheduling of cloud services, *Scalable Computing: Practice and Experience* 13(3).

Pecora, F. and Cesta, A., (2007). Dcop for smart homes: A case study, *Computational Intelligence* 23(4), 395–419.

Pedrasa, M. A. A., Spooner, T. D. and MacGill, I. F., (2010). Coordinated scheduling of residential distributed energy resources to optimize smart home energy services, *Smart Grid, IEEE Transactions on* 1(2), 134–143.

Pelletier, M.-P., Trpanier, M. and Morency, C., (2011). Smart card data use in public transit: A literature review, *Transportation Research Part C: Emerging Technologies* 19(4), 557 – 568. URL: *http://www.sciencedirect.com/science/article/pii/S0968090X1000166X*

Perdikeas, M., Zahariadis, T. and Plaza, P., (2011). The beywatch conceptual model for demand-side management, *in Energy-Efficient Computing and Networking*, Springer, pp. 177–186.

Petersen, K., Kleiner, A. and von Stryk, O., (2013). Fast task-sequence allocation for heterogeneous robot teams with a human in the loop, *in Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, IEEE, pp. 1648–1655.

Preuveneers, D. and Novais, P., (2012). A survey of software engineering best practices for the development of smart applications in ambient intelligence, *Journal of Ambient Intelligence and Smart Environments* 4(3), 149–162.

Ran, Y., Roos, N. and van den Herik, J., (2002). Approaches to find a near-minimal change solution for dynamic CSPs, *in Fourth international workshop on integration of AI and OR techniques in constraint programming for combinatorial optimisation problems*, pp. 373–387.

Reinisch, C., Kofler, M. and Kastner, W., (2010). Thinkhome: A smart home as digital ecosystem, *in Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on*, IEEE, pp. 256–261.

Rodden, T., Cheverst, K., Davies, K. and Dix, A., (1998). Exploiting context in hci design for mobile systems, *in Workshop on human computer interaction with mobile devices*, Citeseer, pp. 21–22.

Roos, N., Ran, Y. and Van Den Herik, J., (2000). Combining local search and constraint propagation to find a minimal change solution for a dynamic csp, *in Artificial Intelligence: Methodology, Systems, and Applications*, Springer, pp. 272–282.

Russell, S. J. and Norvig, P., (2002). *Artificial Intelligence: A Modern Approach, 2nd Ed.*, Prentice Hall, Englewood Cliffs, NJ.

Ryan, N. S., Pascoe, J. and Morse, D. R., (1998). Enhanced reality fieldwork: the context-aware archaeological assistant, *in Computer applications in archaeology*, Tempus Reparatum.

Schiex, T. and Verfaillie, G., (1994). Nogood recording for static and dynamic constraint satisfaction problems, *Int. Journal of Artificial Intelligence Tools* 3-2, 187–207.

Schilit, B., Adams, N. and Want, R., (1994). Context-aware computing applications, *in Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, IEEE, pp. 85–90.

Spanoudakis, N. I. and Moraitis, P., (2006). Agent-based architecture in an ambient intelligence context., *in EUMAS*.

Taqqali, W. M. and Abdulaziz, N., (2010). Smart grid and demand response technology, *in Energy Conference and Exhibition (EnergyCon), 2010 IEEE International*, IEEE, pp. 710–715.

Truong, H.-L. and Dustdar, S., (2009). A survey on context-aware web service systems, *International Journal of Web Information Systems* 5(1), 5–31.

Verfaillie, G. and Jussien, N., (2005). Constraint solving in uncertain and dynamic environments: A survey, *Constraints* 10(3), 253–281.

Verfaillie, G. and Schiex, T., (1994). Solution reuse in dynamic constraint satisfaction problems, *in Proc. of the National Conference on Artificial Intelligence*, pp. 307–312.

Videla, A. and Williams, J. J., (2012). *RabbitMQ in action*, Manning.

Wahl, F., Milenkovic, M. and Amft, O., (2012). A distributed PIR-based approach for estimating people count in office environments, *in Int. Conf. Computational Science and Engineering (CSE)*, IEEE, pp. 640–647.

Want, R., Hopper, A., Falcão, V. and Gibbons, J., (1992). The active badge location system, *ACM Transactions on Information Systems (TOIS)* 10(1), 91–102.

Weiser, M., (1991). The computer for the 21st century, *Scientific american* 265(3), 94–104.

Weiss, M. and Guinard, D., (2010). Increasing energy awareness through web-enabled power outlets, *in Proceedings of the 9th International Conference on Mobile and Ubiquitous Multimedia*, ACM, p. 20.

White, T., (2012). *Hadoop: The definitive guide*, O'Reilly Media.

Xiong, G., Chen, C., Kishore, S. and Yener, A., (2011). Smart (in-home) power scheduling for demand response on the smart grid, *in Innovative Smart Grid Technologies (ISGT), 2011 IEEE PES*, IEEE, pp. 1–7.

Xu, C. and Cheung, S. C., (2005). Inconsistency detection and resolution for context-aware middleware support, *in Proceedings of the Joint 10th European software engineering conference and 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM, pp. 336–345.

Xu, C., Cheung, S. C. and Chan, W. K., (2006). Incremental consistency checking for pervasive context, *in ICSE '06: Proceedings of the 28th international conference on Software engineering*, ACM, pp. 292–301.

Xu, C., Cheung, S. C., Chan, W. K. and Ye, C., (2008). Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications, *in Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, IEEE Computer Society, pp. 713–721.

Xu, C., Cheung, S. C., Chan, W. K. and Ye, C., (2010). Partial constraint checking for context consistency in pervasive computing, *ACM Trans. Softw. Eng. Methodol.* 19(3), 1–61.

Ye, J. and Dobson, S., (2010). Exploring semantics in activity recognition using context lattices, *Journal of Ambient Intelligence and Smart Environments* 2(4), 389–407.

Youngblood, G. M., Cook, D. J. and Holder, L. B., (2004). The MavHome architecture, *Department of Computer Science and Engineering University of Texas at Arlington, Techinal Report,* vol. 39.

Zhang, D., Gu, T. and Wang, X., (2005). Enabling context-aware smart home with semantic web technologies, *International Journal of Human-friendly Welfare Robotic Systems* 6(4), 12–20.

Samenvatting

Slimme huizen, en in het algemeen, andere soorten slimme omgevingen kunnen worden gedefinieerd door verschillende belangrijke karakteristieken. De belangrijkste hiervan is ongetwijfeld de mogelijkheid om omgevingsbewust te zijn, om de fysieke omgeving te ervaren en om de context van de huidige situatie te begrijpen. Slimme omgevingen zouden in staat moeten zijn om met deze informatie te kunnen redeneren en waardevolle kennis te kunnen afleiden. Daarnaast zullen ze de mogelijkheid moeten hebben om intelligent te reageren in reactie op veranderende situaties, volgens bepaalde doelstellingen. Slimme omgevingen zijn vaak *ubiquitous*, wat betekent dat hun capaciteiten voor waarnemen en handelen berusten op apparaten die zijn ingebed in de fysieke wereld.

Er zijn verschillende criteria op basis waarvan de intelligentie van slimme omgevingen kan worden beoordeeld. De meeste slimme omgevingen zijn ontworpen om het comfort en de kwaliteit van leven van hun gebruikers, bv. de bewoners van een gebouw, te verhogen. Het automatiseren van omringende apparaten geschied normaliter ten behoeve van dit doel, bijvoorbeeld door de huidige doelen en problemen te begrijpen en door acties te ondernemen om deze op te lossen. In de meeste gevallen, echter, zou dit niet mogen leiden tot situaties waar gebruikers niet in staat zijn om de beslissingen van het systeem te overschrijven, omdat dit niet alleen het niveau van comfort sterk verlaagt, maar ook omdat dit gevaarlijk kan zijn in sommige onvoorziene situaties. Daarom is tevens de mogelijkheid van de gebruikers om de slimme omgeving te besturen een belangrijk criterium. Veel slimme omgevingen zijn ontworpen met name om ouderen en gehandicapten te helpen, en om zodoende het gezond ouder worden te ondersteunen. En, natuurlijk, de

stijgende energieprijzen en het gebruik van hernieuwbare energiebronnen brengen het onderwerp van energiebewustzijn en energiebesparingen in slimme omgevingen ter tafel.

De meeste van de huidige commerciële slimme omgevingsproducten presenteren slechts gedeeltelijke oplossingen, zoals automatische verlichting of energiebewustzijn. Verschillende factoren vertragen de commercialisering van volledig slimme huizen, waaronder de noodzaak om de oplossing op iedere nieuwe locatie opnieuw zeer nauwkeurig af te stellen, de inspanningen rondom de integratie en coördinatie van verschillende componenten, handelingen om een consistent model over verschillende subsystemen van verschillende bronnen samen te stellen, enzovoorts. Samenvattend, de grote hoeveelheid aan inspanningen die benodigd zijn om de oplossing van een locatie naar een andere te verplaatsen hindert de mogelijkheden voor het stroomlijnen van de uitrol.

In dit proefschrift bespreken we en geven we antwoord op een aantal belangrijke onderzoeksvraagstukken voor huidige pervasieve systemen, slimme omgevingen in het bijzonder.

***RQ1**. Wat zijn de overeenkomsten in het ontwerp en het ontwikkelproces van een slimme omgeving? Valt enig patroon af te leiden van de technische architectuur van deze systemen? Hoe kan het proces gestroomlijnd en vereenvoudigd worden? Welke kennis van bestaande projecten kan worden hergebruikt in nieuwe projecten?*

Hoofdstuk 3 van het proefschrift analyseert huidige en eerdere systemen en projecten voor slimme huizen. Het toont aan dat veel voorkomende patronen zich voordoen in architecturen of tijdens de constructie van dergelijke systemen. Ondanks het feit dat veel projecten dergelijke architecturen van de grond af ontwerpen, kan de kennis van projecten uit het verleden hergebruikt worden om het ontwerpen van de architectuur te vergemakkelijken, of, in sommige gevallen kan een project zelfs in zijn geheel bestaande architectuuroplossingen hergebruiken.

***RQ2**. Wat is een effectieve aanpak om een redeneringsmotor voor slimme omgevingen te ontwerpen die aan alle belangrijke vereisten voldoet (zoals schaalbaarheid, robuustheid, dynamische adaptatie, computationele efficiëntie, real-time antwoorden, enzovoorts)? Zijn er bepaalde specifieke structuren of onderscheidende functionaliteiten van slimme omgevingsdomeinen? Als dit het geval is, is het mogelijk om deze specifieke structuur te benutten om de prestaties en/of de redeneringsmogelijkheden van een redeneringsmotor werkend met deze domeinen te verbeteren?*

Hoofdstuk 4 beschrijft een redeneringscomponent dat gebaseerd is op principes van dynamische randvoorwaardenvervulling. Het hoofdstuk laat zien waarom het

randvoorwaardenvervullingsmodel goed is voor het modelleren van besluitvorming in slimme omgevingen die moeten reageren op veranderende omgevingsomstandigheden. Het hoofdstuk toont tevens aan dat rechtstreekse representatie van het randvoorwaardenvervullingsprobleem leidt tot een groot aantal excessieve berekeningen. De belangrijkste bijdrage van het hoofdstuk is daarom het voorstellen van een methode om een redeneringstaak als een randvoorwaardenvervullingsprobleem te modelleren op een manier waarop niet-noodzakelijke herberekeningen worden te voorkomen wanneer nieuwe gebeurtenissen plaatshebben in de omgeving.

***RQ3**. Hoe kan het effect van sensorfouten voor wat betreft de besluitvorming worden geminimaliseerd? Kan een redeneringsmotor functioneren met onvolledige en/of tegenstrijdige sensorgegevens? Indien er geen definitief antwoord bestaat over welke gegevens incorrect zijn, kan het systeem correct functioneren in de aanwezigheid van tegenstrijdige gegevens?*

Hoofdstuk 5 beschrijft een manier om onjuiste meetwaarden op een probabilistische wijze te detecteren door het gebruik van onderlinge afhankelijkheidsregels tussen de sensorvariabelen en de datastructuur van een contextconsistentiediagram (CCD) als een manier om de meest waarschijnlijke situatie van de huidige omgeving te bepalen in het geval dat de meetwaarden onduidelijke, onvolledige, of tegenstrijdige informatie verschaffen. Hoofdstuk 6 beschrijft een manier om een klassieke CCD te reduceren waarbij wel de mogelijkheid om de meest waarschijnlijke situatie te berekenen wordt behouden.

***RQ4**. Hoe kan een slim systeem het bestaan van verschillende energieleveranciers gebruiken om de energiekosten in de tijd te minimaliseren? Beïnvloedt dit slimme systeem het totale energieverbruik? Welke informatie zou in dit geval beschikbaar moeten zijn voor een redeneringsmotor, en hoe kan deze optimaal gebruikt worden?*

Hoofdstuk 7 behandelt het inplannen van apparaten in de tijd voor huizen die zijn aangesloten op een *smart grid*. In een dergelijke omgeving, waar goedkope energie beperkt is en de prijs verandert in de tijd, is het belangrijk om apparaten die grotendeels onafhankelijk zijn van menselijke interactie in te plannen op een manier waarmee gelijktijdig gebruik wordt verminderd en het werk meestal verplaatst wordt naar daluren. De planner is gebouwd om domeinonafhankelijk en herbruikbaar te zijn in andere domeinen die een soortgelijk beleid bevatten.